DATAPOINT

DASL™

User's Guide

50807

NOVEMBER 14, 1984

Document No. 50807-01. 11/84

Preface

The DASL USER's GUIDE, Vol. IV, is another reference in the DASL library. Vol. IV describes the Common User Functions.

This work is a DATEC production. CUFs was written on a DATAPOINT processor using an experimental writing tool. Then, the masters for printing were produced on a DATAPOINT 9660 Laser Printer.

Please forward your comments on this document to:

DATEC Publications
DATAPOINT Corporation
9725 Datapoint Dr. MS T-72
San Antonio, Texas 78284

TABLE OF CONTENTS

6. MESSAGE HANDLING CUFS

7. LIST PROCESSING CUFS

8. PROGRAM STRUCTURING CUFS

9. SCHEDULER CUFS

10. MISCELLANEOUS CUFS

Chapter 1.
INTRODUCTION
OVERVIEW

About this document

This document describes the Common User Functions. It contains general information about the Common User Functions, as well as a detailed description of each individual Common User Function.

Intended audience

This document is intended for programmers writing software in DASL on the RMS operating system.

It contains code segments of DASL programs that reference the NOSL I/O package, the D$IO package, and the RMS operating system.

How this document is organized

This document is divided into ten chapters. Chapters 3 through 10 contain detailed descriptions of the Common User Functions which have been categorized by programming function.

Chapter	*Content*
1	Introduction to the Common User Functions.
2	Procedures for including a Common User Function into your DASL program.
3	Memory allocation Common User Functions.
4	Character and string Common User Functions.
5	Input/output Common User Functions.
6	Message handling Common User Functions.
7	List processing Common User Functions.
8	Program structuring Common User Functions.
9	Scheduler Common User Functions.
10	Miscellaneous Common User Functions.

INTRODUCTION TO COMMON USER FUNCTIONS

Introduction

You can probably remember an occasion when you needed to write code that another programmer had already written. But because that code was undocumented or buried deep within an unfamiliar program, you probably ended up writing your own version.

In response to this problem, a collection of useful, efficient, well written functions have been assembled and packaged as the Common User Functions.

Definition of a Common User Function

A Common User Function is

- a macro,
- a function, or
- a set of related macros and functions

specifically designed for solving a DASL programming problem.

Designed to run efficiently

Each Common User Function has been designed to run efficiently. Several functions take advantage of specific machine instructions to attain maximum efficiency.

Common User Functions are also called "CUFs"

The Common User Functions are commonly referred to by the acronym, "CUFs". "CUFs" will be used throughout the remainder of this document.

Anatomy of a CUF

Each CUF may contain one or more of the following DASL elements:

DASL Element	*Description*
Type definition	Type definitions define a variable structure needed by a CUF.
Variable	Variables are flags, counters, structures, etc... that • must be declared by you prior to calling a CUF, or • are provided for you to examine the status or result of a CUF.
Macro	Macros • interface you to a CUF function, or • are changed into a problem solving statement or group of statements. Note: Macros that are called like functions will be referred to as functions throughout the remainder of this document.
Function	Functions solve part or all of a common programming problem.

Examples of CUFs

Some examples of CUFs are

- a macro that determines the maximum of two values,
- a function that makes a copy of a string, and
- a set of functions and macros that dynamically allocate memory.

Chapter 2.
INCLUDING A CUF
OVERVIEW

Introduction

This chapter contains the procedures for including a CUF into your DASL program.

Overview of the procedure

The procedures for including a CUF are outlined in the following table. A description of each procedure in this table is included in this chapter.

Task	*Action*
1	Include the CUF definition file into your DASL program.
2	Declare any variables needed by the CUF in your DASL text.
3	Initialize any variables needed by the CUF in your DASL text.
4	Edit the CUF function call into your DASL text.
5	Include the CUF relocatable library in your LINK directives.

INCLUDE CUF DEFINITION FILE

Introduction

Each individual CUF has a corresponding CUF definition file which must be included in the INCLUDE portion of your DASL program.

Task	*Action*
1	Include the CUF definition file into your DASL program.

Description of a CUF definition file

A CUF definition file is a text file that contains definitions and external references that are needed to reference the CUF. Its filename is made from the name of the CUF with a "/DEFS" extension.

Example: The $SCAN CUF filename is $SCAN/DEFS.

Example

The following program segment includes the $SCAN CUF definition file.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE($SCAN/DEFS)                    /* Include $SCAN CUF */
:
: (rest of the program)
```

DECLARE CUF VARIABLES

Introduction

Several CUFs require you to declare variables of certain types to be passed as parameters to the CUF functions or returned as results of the CUF functions.

Task	*Action*
2	Declare any variables needed by the CUF in your DASL text.

Example

The $SCAN function of the $SCAN CUF requires you to pass the address of the first character of a string, a character to scan for in the string, and the number of characters to scan in the string. The following program segment declares the variables needed for the call to $SCAN.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE($SCAN/DEFS)              /* Include $SCAN CUF */

ENTRY MAIN () :=
VAR   n BYTE;
      string [80] CHAR;
      scanChar CHAR;
{
   scanChar := '?';
   n := n$read(n$WSIn, &string[0], SIZEOF string);
   IF $SCAN(&string[0], scanChar, n) = n THEN
      error('Scan character not found.');
   :
   : (rest of the program)
```

INITIALIZE CUF VARIABLES

Introduction

Several CUFs require you to initialize variables prior to calling the functions.

Task	*Action*
3	Initialize any variables needed by the CUF in your DASL text.

Example

The $SCAN function requires you to pass in an initialized string, character, and number. The following program segment initializes these variables.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE($SCAN/DEFS)                    /* Include $SCAN CUF */

ENTRY MAIN () :=
VAR   n BYTE;
      string [80] CHAR;
      scanChar CHAR;
{
   scanChar := '?';
   n := n$read(n$WSIn, &string[0], SIZEOF string);
   IF $SCAN(&string[0], scanChar, n) = n THEN
      error('Scan character not found.');
   :
   : (rest of the program)
```

CALL CUF FUNCTION

Introduction

After you have declared and initialized any variables required by the CUF function, the function call must be edited into your DASL text.

Task	*Action*
4	Edit the CUF function call into your DASL text.

Example

The following program segment calls the $SCAN function of the $SCAN CUF.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE($SCAN/DEFS)              /* Include $SCAN CUF */

ENTRY MAIN () :=
VAR   n BYTE;
      string [80] CHAR;
      scanChar CHAR;
{
   scanChar := '?';
   n := n$read(n$WSIn, &string[0], SIZEOF string);
   IF $SCAN(&string[0], scanChar, n) = n THEN
      error('Scan character not found.');
   :
   : (rest of the program)
```

INCLUDE CUF RELOCATABLE LIBRARY

Introduction

All of the individual CUF's relocatable object files are combined into a single relocatable library which must be included by the LIBRARY directive in LINK.

Task	*Action*
5	Include the CUF relocatable library in your LINK directives.

Selecting the correct CUFs library

There are several copies of the CUFs library that support three of the DATAPOINT instruction sets. Use this table to select the proper CUFs library for your instruction set.

If you are writing for the...	*then use...*
5500 instruction set	$CUFS/REL5
6600 instruction set	$CUFS/REL
16000 instruction set	$CUFS/RL16

Example

If you had selected $CUFS/REL, your LINK directives might look like this:

```
SEGMENT MYPROG
INCLUDE DASLASM
INCLUDE D$LIB.D$START
LIBRARY N$
LIBRARY $CUFS
LIBRARY D$LIB
LIBRARY RMSUFRS
*
```

Chapter 3.
MEMORY ALLOCATION CUFS
OVERVIEW

Introduction

This chapter contains a description of the two memory allocation CUFs, $ALLOC and $BUDDY.

The memory allocation CUFs obtain memory from RMS that can be dynamically allocated and deallocated by your program. The memory maintained by the memory allocation CUFs is referred to as the "memory pool".

Selecting a CUF

Use the following table to select a memory allocation CUF.

In addition to allocating memory, if you want to...	*then use...*	*page*
• find additional memory when the memory pool has been exhausted, • add memory to the memory pool, or • add the unused memory above the top of your program to the memory pool	$ALLOC.	3-2
• allocate aligned memory, up to 32K at a time, or • use the same memory allocation CUF as the NOSL I/O package	$BUDDY.	3-12

$ALLOC

Introduction

The $ALLOC CUF is a dynamic memory allocation package with supporting functions that can be used to

- find additional memory when the memory pool has been exhausted,
- add memory to the memory pool, and
- add the unused memory above the top of your program to the memory pool.

$ALLOC and RMS

The $ALLOC CUF allocates memory from RMS and manages the memory internally in a memory pool. The $ALLOC CUF does not provide a facility to return the memory to RMS.

$ALLOC variables and functions

Use the following table to select a $ALLOC variable or function.

If you want to...	*then see...*	*page*
determine the amount of memory allocated up till now	$MAXALLOC.	3-3
allocate memory from the memory pool	$ALLOC, or $NEW.	3-4 3-5
return memory to the memory pool	$FREE, or $DISPOSE.	3-6 3-7
allocate memory from RMS	$MOREMEM.	3-8
add memory to the memory pool	$MEMFREE.	3-10
add high memory to the memory pool	$TOPFREE.	3-11

$MAXALLOC variable

$MAXALLOC is declared as part of the $ALLOC CUF and is available for you to examine.

Variable Definition	*Description*
$MAXALLOC UNSIGNED;	Maximum amount of dynamic memory allocated from the memory pool up to now.

$ALLOC function

$ALLOC allocates memory from the memory pool.

Function Syntax	*Input*	*Output*
$ALLOC (size UNSIGNED	Number of bytes to allocate.	
) ^ BYTE;		Points to the allocated memory. Note: Equals NIL if there's no more memory available.

Example:

```
function () :=
VAR   ptr ^ BYTE;
{
   ptr := $ALLOC(4096);
```

$NEW function

$NEW allocates enough memory for whatever the input variable is pointing to.

Function Syntax	*Input*	*Output*
$NEW (ptr ^ Anything);	Pointer to any type.	Points to the allocated variable of that type. Note: Equals NIL if there's no more memory available.

Example:

```
function () :=
VAR   ptr ^ $SFENT;
{
   $NEW(ptr);
```

$FREE function

$FREE returns memory to the memory pool.

Function Syntax	*Input*	*Output*
$FREE (ptr ^ BYTE);	Pointer to the first byte of memory to return. Note: Must point to memory that was previously allocated by $ALLOC or $NEW.	

Example:

```
function () :=
VAR   ptr ^ BYTE;
{
   ptr := $ALLOC(4096);
   $FREE(ptr);
```

$DISPOSE function

$DISPOSE returns the memory used by whatever the input variable is pointing to back to the memory pool.

Function Syntax	*Input*	*Output*
$DISPOSE (ptr ^ Anything);	Pointer to a variable of any type. Note: Must point to a variable that was previously allocated by $ALLOC or $NEW.	

Example:

```
function () :=
VAR   ptr ^ $SFENT;
{
   $NEW(ptr);
   $DISPOSE(ptr);
```

$MOREMEM function

$MOREMEM allocates additional memory from RMS when the memory pool has been exhausted. $MOREMEM is

- automatically called by $ALLOC and $FREE when the memory pool has been exhausted and
- automatically calls $MEMFREE if it is successful in allocating additional memory.

Function Syntax	*Input*	*Output*
$MOREMEM (size UNSIGNED	Amount of memory needed.	
) BOOLEAN;		TRUE if additional memory was found FALSE otherwise.

Note: There are benefits for replacing this function with your own version. Two replacements are described below.

- Define $MOREMEM to terminate program execution with an error message if it cannot get additional memory. You can avoid having to check each $ALLOC call for a NIL return by defining $MOREMEM this way.
- Define $MOREMEM to $MEMFREE any buffers or tables that are no longer used by your program to the memory pool and return TRUE.

$MOREMEM function (continued)

Example: The following program segment replaces the CUF $MOREMEM function with the first of the replacement functions described on the previous page.

Note: $MOREMEM must be declared "ENTRY".

```
ENTRY $MOREMEM (size UNSIGNED) BOOLEAN :=
VAR   msn BYTE;
      i, n BYTE;
      psk BYTE;
{
   RESULT := FALSE;
   msn := 0;
   i := 0;
   n := (size + 4095) / 4096;
   LOOP {
    WHILE i < n & msn < 16;
      IF $MEMKEY(msn++ << 4, &psk) && D$CFLAG
        & $ERRC.$CODE = 1 THEN i++
       ELSE i := 0;
      };
   IF i = n THEN {
      i := 0;
      LOOP {
       WHILE i < n
        & ~($$MEMGET(&psk) && D$CFLAG)
        & ~($MEMMAP(--msn << 4, psk) && D$CFLAG);
         i++;
         };
      IF i = n THEN {
         $MEMFREE(<^ BYTE>(msn << 12), 4096 * n);
         RESULT := TRUE;
         };
      };
   IF ~ RESULT THEN {
      display('FATAL ERROR: No more memory.');
      $ERROR();
      };
   };
```

$MEMFREE function

$MEMFREE adds memory to the memory pool.

Note: You don't need to call this function unless

- you are setting up your own memory pool, or
- you replaced the $MOREMEM function with your own version and are giving additional memory to $ALLOC.

Function Syntax	*Input*	*Output*
$MEMFREE (ptr ^ BYTE,	Address of the first byte in memory to add to the memory pool.	
size UNSIGNED);	Number of bytes to add to the memory pool.	

Example:

```
function () :=
VAR   STATIC memPool [500] BYTE;
{
   $MEMFREE(&memPool[0], SIZEOF memPool);
```

$TOPFREE function

$TOPFREE adds the unused memory above the top of your program to the memory pool.

You must define $LOADTOP as the "next available byte" address label of the SEGMENT directive in the linkage phase of the DASL compile process.

Example: Your LINK directives might look like this:

```
SEGMENT MYPROG,,$LOADTOP
INCLUDE DASLASM
INCLUDE D$LIB.D$START
LIBRARY N$
LIBRARY $CUFS
LIBRARY D$LIB
LIBRARY RMSUFRS
*
```

Function Syntax	*Input*	*Output*
$TOPFREE ( );		

Example:

```
function () :=
{
   $TOPFREE();
```

$BUDDY

Introduction

The $BUDDY CUF is a dynamic memory allocation package that allocates aligned memory based upon the amount of memory requested from the memory pool.

Aligned memory

Memory allocated from $BUDDY is aligned on the nearest 2^n boundary, where 2^n is the lowest power of 2 greater than or equal to the amount of memory requested. For example:

- a request for 256 bytes will return memory aligned on a page boundary that may be used for a $PFDB buffer, and
- a request for 4096 bytes will return memory aligned on a sector boundary which can be mapped in and out of logical space.

$BUDDY and RMS

The $BUDDY CUF allocates memory from RMS and manages the memory internally in a memory pool. The $BUDDY CUF does not provide a facility to return the memory to RMS.

$BUDDY functions

Use the following table to select a $BUDDY function.

If you want to...	*then see...*	*page*
allocate memory from memory pool	$alloc, or $new.	3-13 3-14
return memory to memory pool	$free, or $dispose.	3-15 3-16

$alloc function

$alloc allocates aligned memory from the memory pool.

Function Syntax	*Input*	*Output*
$alloc (size UNSIGNED	Number of bytes to allocate.	
) ^ BYTE;		Points to the allocated memory. Note: Equals NIL if there's no more memory available.

Example:

```
function () :=
VAR   ptr ^ BYTE;
{
   ptr := $alloc(4096);
```

$new function

$new allocates aligned memory for whatever type the input variable is pointing to.

Function Syntax	*Input*	*Output*
$new (ptr ^ Anything);	Pointer to any type.	Points to the allocated variable of that type. Note: Equals NIL if there's no more memory available.

Example:

```
function () :=
VAR   ptr ^ $SFENT;
{
   $new(ptr);
```

$free function

$free returns memory to the memory pool.

Function Syntax	*Input*	*Output*
$free (ptr ^ BYTE,	Pointer to the first byte of memory to return. Note: Must point to memory that was previously allocated by $alloc or $new.	
size UNSIGNED);	Number of bytes of memory to return to the memory pool. Note: The number of bytes must be the same as the number originally allocated.	

Example:

```
function () :=
VAR   ptr ^ BYTE;
{
   ptr := $alloc(4096);
   $free(ptr, 4096);
```

$dispose function

$dispose returns the memory used by whatever the input variable is pointing to back to the memory pool.

Function Syntax	*Input*	*Output*
$dispose (ptr ^ Anything);	Pointer to a variable of any type. Note: Must point to a variable that was previously allocated by $alloc or $new.	

Example:

```
function () :=
VAR   ptr ^ $SFENT;
{
   $new(ptr);
   $dispose(ptr);
```

Chapter 4.
CHARACTER AND STRING CUFS
OVERVIEW

Introduction

This chapter describes the character and string handling CUFs.

The character and string handling CUFs are used to search, compare, convert, and clear characters and strings.

Selecting a CUF

Use the the following to select a character and string handling CUF.

If you want to...	*then use...*	*page*
• perform a byte-by-byte translation on an input string, or • look for the first occurrence in a string for one of a number of possible characters	$CONVERT.	4-2
initialize a table, array, etc... with a certain character	$FILL.	4-5
compare two strings for a mismatch	$MATCH.	4-6
scan a string for a character	$SCAN.	4-7
search a string for an instance of another string	$SEARCH.	4-10
define, compare, or copy strings that are part of a type that also includes the string length	$STRING or $STRCMP.	4-13 4-17

$CONVERT

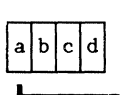
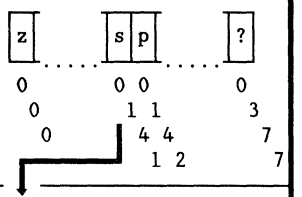
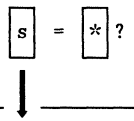
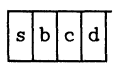
Introduction

The $CONVERT CUF is a single function which performs a byte-by-byte translation on the input string until

- the termination character is found in the string, or
- the maximum number of characters to translate is reached.

Byte-by-byte translation process

The byte-by-byte translation process is shown in the following diagram.

$CONVERT...	*Example*
takes the byte value of the character in the input string,	 byte value of "a" is 0141
uses that value as an index into the translation table that you provide,	
stops if the character from the translation table is equal to the termination character,	 example termination character is "*"
and replaces the string character with the character from the translation table.	 the "a" in the string is replaced by an "s".

$CONVERT function

$CONVERT performs a byte-by-byte translation on the input string looking for a termination character.

Function Syntax	*Input*	*Output*
$CONVERT (str ^ CHAR,	Address of the first character of a string.	
len UNSIGNED,	Number of characters to translate in the string.	
tablePtr ^ [256] CHAR,	Address of a translation table. Note: The translation table must be page aligned for the 55/6600 instruction set.	
term CHAR	Terminate translation if this character is found in the string. Note: This character cannot be 0 on the 55/6600 instruction set.	
) UNSIGNED;		Number of characters translated. Note: Does not include "term" if found.

$CONVERT function (continued)

Example: The following program segment demonstrates how to use the $CONVERT CUF to look for the first occurrence of any of several characters in an input string. Note how the translation table is initialized to translate all characters except 'b' into themselves. $CONVERT will terminate if it finds either 'b' or 'c' in the input string.

```
function () :=
VAR   trans ^ [256] CHAR;
      i BYTE;
      STATIC str [4] CHAR := 'abcd';
{
   $new(trans);  /* allocate aligned trans table */
   i := 0;
   LOOP {
      trans^[i] := i;
    WHILE i++ ~= 255;
      };
   trans^['b'] := 'c';
   i := $CONVERT(&str[0], 4, trans, 'c');
```

$FILL

Introduction

The $FILL CUF is a single function which initializes memory.

$FILL function

$FILL initializes strings, arrays, tables, structures, etc... to a specified character.

Function Syntax	*Input*	*Output*
$FILL (p ^ BYTE,	Address of the first byte to initialize.	
size UNSIGNED,	Number of bytes to initialize.	
filler BYTE);	Value to store into each byte.	

Example:

```
function () :=
VAR   a [1000] CHAR;
{
   $FILL(&a[0], SIZEOF a, '0');
```

$MATCH

Introduction

The $MATCH CUF is a single function which compares two strings for a mismatch.

$MATCH function

$MATCH compares two strings for a mismatch.

Function Syntax	*Input*	*Output*
$MATCH (p ^ BYTE,	Address of the first byte of the first string to compare.	
q ^ BYTE,	Address of the first byte of the second string to compare.	
len UNSIGNED	Number of bytes to compare.	
) UNSIGNED;		Number of bytes that match.

Example:

```
function () :=
VAR    s [256] CHAR;
       STATIC p [8] CHAR := 'PASSWORD';
{
   n$read(n$WSIn, &s[0], SIZEOF p+1);
   IF $MATCH(&p[0], &s[0], 8) = 8 THEN matched();
```

$SCAN

Introduction

The $SCAN CUF is a character scanning package which scans

- forward or
- backward

through a string for a character.

$SCAN functions

Use the following table to select a $SCAN function.

If you want to...	*then see...*	*page*
scan forward for character	$SCAN.	4-8
scan backward for character	$SCANR.	4-9

$SCAN function

$SCAN scans forward through a string for a character.

Function Syntax	*Input*	*Output*
$SCAN (str ^ CHAR,	Address of the first character in the input string.	
key CHAR,	Character to look for in the string.	
len UNSIGNED	Number of characters in the string to scan.	
) UNSIGNED;		Number of characters that were skipped before the character was found.

Example:

```
function () :=
VAR   n BYTE;
      s [80] CHAR;
{
   n := n$read(n$WSIn, &s[0], SIZEOF s);
   IF $SCAN(&s[0], '?', n) = n THEN
      error('Invalid input string');
```

$SCANR function

$SCANR scans backward through a string for a character.

Function Syntax	*Input*	*Output*
$SCANR (str ^ CHAR,	Address of the last character in the input string.	
key CHAR,	Character to look for in the string.	
len UNSIGNED	Number of characters in the string to scan.	
) UNSIGNED;		Number of characters that were skipped before the character was found.

Example:

```
function () :=
VAR   n BYTE;
      s [80] CHAR;
{
   n := n$read(n$WSIn, &s[0], SIZEOF s);
   n := $SCANR(&s[n-1], 'A', n);
```

$SEARCH

Introduction

The $SEARCH CUF is a string searching package which searches

- forward or
- backward

through a string for an instance of another string.

$SEARCH functions

Use the following table to select a $SEARCH function.

If you want to...	*then see...*	*page*
search forward for string	$SEARCH.	4-11
search backward for string	$SEARCHR.	4-12

$SEARCH function

$SEARCH searches forward through a string for an instance of another string.

Function Syntax	*Input*	*Output*
$SEARCH (key ^ CHAR,	Address of the first character of the string you are searching for.	
keyLen UNSIGNED,	Number of characters in the string you are searching for.	
str ^ CHAR,	Address of the first character of the string you are searching.	
strLen UNSIGNED	Number of characters in the string you are searching.	
) UNSIGNED;		Number of characters that were skipped before the string was found.

Example:

```
function () :=
VAR   n BYTE;
      s [80] CHAR;
      STATIC key [] CHAR := 'HELLO';
{
   n := n$read(n$WSIn, &s[0], SIZEOF s);
   n := $SEARCH(&key[0], SIZEOF key, &s[0], n-1);
```

$SEARCHR function

$SEARCHR searches backward through a string for an instance of another string.

Function Syntax	*Input*	*Output*
$SEARCHR (key ^ CHAR,	Address of the last character of the string you are searching for.	
keyLen UNSIGNED,	Number of characters in the string you are searching for.	
str ^ CHAR,	Address of the last character of the string you are searching.	
strLen UNSIGNED	Number of characters in the string you are searching.	
) UNSIGNED;		Number of characters that were skipped before the string was found.

Example:

```
function () :=
VAR   n BYTE;
      s [80] CHAR;
      STATIC key [] CHAR := 'HELLO';
{
   n := n$read(n$WSIn, &s[0], SIZEOF s);
   n := $SEARCHR(&key[4], SIZEOF key, &s[n-1], n);
```

$STRING

Introduction

The $STRING CUF is a string manipulation package which contains a macro and functions to

- initialize,
- compare, and
- copy

strings which are declared as part of a type, $String, which also includes their length.

$STRING types, macros, and functions

Use the following table to select a $STRING type, macro, or function.

If you want to...	*then see...*	*page*
define a string variable	$String.	4-14
initialize a $String variable	$str.	4-14
check two strings for equality	$stringEqual.	4-16
copy a string	$stringCopy.	4-16

$String type

$String is defined as follows:

Type Definition	*Description*
TYPDEF $String STRUCT { ptr ^ CHAR;	Pointer to a string.
len UNSIGNED; };	Length of the pointed string.

$str macro

$str is a macro which generates an initializer for variables of type $String.

Function Syntax	*Input*	*Output*
$str (str [any size] CHAR	Any character string.	
) $String;		A $String initializer containing the string and the string length.

Example:

```
function () :=
VAR   STATIC s $String := $str('I am a string');
```

$stringEqual function

$stringEqual compares two strings, described by their respective $String types, to determine if they are equal.

Function Syntax	*Input*	*Output*
$stringEqual (strl ^ $String,	Address of the first string descriptor.	
str2 ^ $String	Address of the second string descriptor.	
) BOOLEAN;		TRUE if the strings are equal in content and length. FALSE otherwise.

Example:

```
function () :=
VAR   STATIC key $String := $str('PASSWORD');
      st $String;
      s [80] CHAR;
{
   st.ptr := &s[0];
   st.len := n$read(n$WSIn, st.ptr, SIZEOF s) - 1;
   IF $stringEqual(&key, &st) THEN equal();
```

$stringCopy function

$stringCopy copies one string, described by the $String type, into another.

Function Syntax	*Input*	*Output*
$stringCopy (from ∧ $String,	Address of the source string descriptor.	
to ∧ $String,	Address of the destination string descriptor.	Describes the copied string. Note: No copy is done if source is larger than the destination.
leftOver ∧ $String	May be NIL. If not it is used as an output variable.	Describes the unused area in the resulting destination string, to∧.
) BOOLEAN;		TRUE if source fits into destination. FALSE otherwise.

Example:

```
function () :=
VAR    STATIC sourcel $String := $str('Source');
       STATIC source2 $String := $str(' String');
       s [80] CHAR;
       STATIC dest $String := $str(s);
       leftOver $String;
{
   IF $stringCopy(&sourcel, &dest, &leftOver)
     & $stringCopy(&source2, &leftOver, NIL) THEN
       concatenatedStrings();
```

$STRCMP

Introduction

The $STRCMP CUF is a single function which compares two strings that are described by the $String type.

Note: $String is declared in the $STRING CUF.

$STRCMP definition file

The $STRCMP definition file must be included after the $STRING definition file.

Example: Your DASL include directives might look like this:

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE($STRING/DEFS)
INCLUDE($STRCMP/DEFS)
:
: (rest of the program)
```

$strCmp function

$strCmp compares two strings described by their respective $String types.

Function Syntax	*Input*	*Output*
$strCmp (s1P ^ $String,	Address of the first string descriptor.	
s2P ^ $String	Address of the second string descriptor.	
) SET ($strL,		The first string is less than the second.
$strS,		The first string is an initial substring of the second.
$strE,		The strings are equal.
$strB,		The second string is an initial substring of the first.
$strG);		The first string is greater than the second.

$strCmp function (continued)

Example: The following program segment compares two strings, key and st, and tests the $strE flag which will be set if the two strings are equivalent in length and content.

```
function () :=
VAR    STATIC key $String := $str('PASSWORD');
       st $String;
       s [80] CHAR;
{
   st.ptr := &s[0];
   st.len := n$read(n$WSIn, st.ptr, SIZEOF s) - 1;
   IF $strCmp(&key, &st) && $strE THEN equal();
```

Chapter 5.
INPUT/OUTPUT CUFS
OVERVIEW

Introduction

This chapter describes the input/output CUFs.

The input/output CUFs are used to handle general keyboard and file I/O.

Selecting a CUF

Use the following table to determine which input/output CUF to use.

If you want to...	*then use...*	*page*
allocate a page aligned buffer and a $PFDB that describes the buffer	$BUFFER.	5-2
open or create a binary or text file under D$IO and automatically allocate buffers and supporting file structures	$EZOPEN.	5-6
read a text file forward and backward and insert and delete lines	$SEQFH.	5-22
obtain a single translated character from the keyboard	$GETKEYCH.	5-13
obtain a single untranslated character from the keyboard	$KEYCH.	5-17

$BUFFER

Introduction

The $BUFFER CUF is a file I/O package that allocates and deallocates a page aligned buffer and a $PFDB (with $PFDBBUFs if necessary) that describe the buffer.

Note: $BUFFER uses the $BUDDY CUF for its memory allocation and deallocation.

$BUFFER types and functions

Use the following table to select a $BUFFER type or function.

If you want to...	*then see...*	*page*
declare a buffer descriptor	$BufDesc.	5-3
allocate a page aligned buffer	$alocBuf.	5-4
free a buffer allocated by $alocBuf	$freeBuf.	5-5

$BufDesc type

$BufDesc is defined as follows:

Type Definition	*Description*
TYPDEF $BufDesc STRUCT { addr ^ BYTE;	Points to the allocated page aligned buffer.
size UNSIGNED;	Size of allocated buffer. Note: Always a multiple of 256.
pfdbP ^ $PFDB; };	Points to the allocated $PFDB.

$alocBuf function

$alocBuf allocates a page aligned buffer and a $PFDB that describes the buffer.

Function Syntax	*Input*	*Output*
$alocBuf (bDP ^ $BufDesc,	Address of a buffer descriptor.	Stores the address, size, and $PFDB of the buffer in bDP^.
size UNSIGNED	Number of bytes to allocate for the file buffer. Note: Number is rounded up to a multiple of 256.	
) BOOLEAN;		TRUE if able to allocate buffer. FALSE otherwise.

Example:

```
function () :=
VAR    bufDesc $BufDesc;
{
   IF $alocBuf(&bufDesc, 4096) THEN allocated();
```

$freeBuf function

$freeBuf frees the buffer and $PFDB allocated by $alocBuf and returns it to the memory manager, $BUDDY.

Function Syntax	*Input*	*Output*
$freeBuf (bDP ^ $BufDesc);	Address of the buffer descriptor which describes the buffer and $PFDB to return. Note: Must point to a buffer descriptor that was previously set by $alocBuf.	

Example:

```
function () :=
VAR     bufDesc $BufDesc;
{
   IF $alocBuf(&bufDesc, 4096) THEN allocated();
   $freeBuf(&bufDesc);
```

$EZOPEN

Introduction

The $EZOPEN CUF is an interface to the D$IO package that manages its own buffer allocation and tables. $EZOPEN contains functions for both binary and text files that can

- open a file for reading,
- prepare a file for writing, and
- close a file.

$EZOPEN functions

Use the following table to select a $EZOPEN function.

If you want to...	*then see...*	*page*
open a text file for reading	$textRead.	5-7
open a text file for writing	$textPrep.	5-8
close a text file	$textClose.	5-9
open a binary file for reading	$binaryRead.	5-10
open a binary file for writing	$binaryPrep.	5-11
close a binary file	$binaryClose.	5-12

$textRead function

$textRead opens a D$IO text file for reading.

Note: $textRead will terminate through $ERMSG if the file does not exist.

Function Syntax	*Input*	*Output*
$textRead (sfent ∧ $SFENT	Address of the file entry table of the file to be opened.	
) ∧ D$FILET;		Address of the D$IO text file. Note: NIL if it can't allocate memory.

Example:

```
function () :=
VAR   inFileET $SFENT;
      inFile ∧ D$FILET;
{
   inFileET.$SFTSFN := 'IN      ';
   inFileET.$SFTNAM := 'MYFILE      ';
   inFileET.$SFTEXT := 'TEXT';
   inFileET.$SFTENV := 'W       ';
   inFile := $textRead(&inFileET);
   IF inFile = NIL THEN
      display('Insufficient memory');
```

$textPrep function

$textPrep creates a D$IO text file for writing.

Note: $textPrep will recreate an existing file.

Function Syntax	*Input*	*Output*
$textPrep (sfent ^ $SFENT	Address of the file entry table of the file to be created.	
) ^ D$FILET;		Address of the D$IO text file. Note: NIL if it it can't allocate memory.

Example:

```
function () :=
VAR   outFileET $SFENT;
      outFile ^ D$FILET;
{
   outFileET.$SFTSFN := 'OUT      ';
   outFileET.$SFTNAM := 'MYFILE       ';
   outFileET.$SFTEXT := 'TEXT';
   outFileET.$SFTENV := 'W        ';
   outFile := $textPrep(&outFileET);
   IF outFile = NIL THEN
      display('Insufficient memory');
```

$textClose function

$textClose closes a D$IO text file and deallocates the memory allocated by $textRead or $textPrep.

Function Syntax	*Input*	*Output*
$textClose (filet ^ D$FILET);	Pointer to an open D$IO text file. Note: Must point to a file that was opened by $textRead or $textPrep.	

Example:

```
function () :=
VAR   outFileET $SFENT;
      outFile ^ D$FILET;
{
   outFileET.$SFTSFN := 'OUT      ';
   outFileET.$SFTNAM := 'MYFILE       ';
   outFileET.$SFTEXT := 'TEXT';
   outFileET.$SFTENV := 'W        ';
   outFile := $textPrep(&outFileET);
   $textClose(outFile);
```

$binaryRead function

$binaryRead opens a D$IO binary file for reading.

Note: $binaryRead will terminate through $ERMSG if the file does not exist.

Function Syntax	*Input*	*Output*
$binaryRead (sfent ^ $SFENT	Address of the file entry table of the file to be opened.	
) ^ D$FILEB;		Address of the D$IO binary file. Note: NIL if it it can't allocate memory.

Example:

```
function () :=
VAR   inFileET $SFENT;
      inFile ^ D$FILEB;
{
   inFileET.$SFTSFN := 'IN      ';
   inFileET.$SFTNAM := 'MYFILE      ';
   inFileET.$SFTEXT := 'REL ';
   inFileET.$SFTENV := 'W       ';
   inFile := $binaryRead(&inFileET);
   IF inFile = NIL THEN
      display('Insufficient memory');
```

$binaryPrep function

$binaryPrep creates a D$IO binary file for writing.

Note: $binaryPrep will recreate an existing file.

Function Syntax	*Input*	*Output*
$binaryPrep (sfent ^ $SFENT,	Address of the file entry table of the file to be created.	
format BYTE	RMS file format. Example: $FFMTBIN	
) ^ D$FILEB;		Address of the D$IO binary file. Note: NIL if it it can't allocate memory.

Example:

```
function () :=
VAR   outFileET $SFENT;
      outFile ^ D$FILEB;
{
   outFileET.$SFTSFN := 'IN      ';
   outFileET.$SFTNAM := 'MYFILE      ';
   outFileET.$SFTEXT := 'REL ';
   outFileET.$SFTENV := 'W       ';
   outFile := $binaryPrep(&outFileET, $FFMTL55);
   IF outFile = NIL THEN
      display('Insufficient memory');
```

$binaryClose function

$binaryClose closes a D$IO binary file and deallocates the memory allocated by $binaryRead or $binaryPrep.

Function Syntax	*Input*	*Output*
$binaryClose (filet ^ D$FILEB);	Pointer to an open D$IO binary file. Note: Must point to a file that was opened by $binaryRead or $binaryPrep.	

Example:

```
function () :=
VAR   outFileET $SFENT;
      outFile ^ D$FILEB;
{
   outFileET.$SFTSFN := 'IN      ';
   outFileET.$SFTNAM := 'MYFILE      ';
   outFileET.$SFTEXT := 'REL ';
   outFileET.$SFTENV := 'W       ';
   outFile := $binaryPrep(&outFileET, $FFMTL55);
   $binaryClose(outFile);
```

$GETKEYCH

Introduction

The $GETKEYCH CUF is a general keyboard I/O package that contains

- definitions for the cursor control keypad,
- a function to obtain one translated character from the keyboard, and
- a function to turn shift inversion on or off.

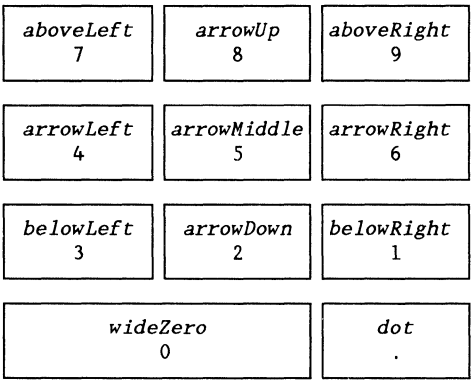
$GETKEYCH definitions and functions

Use the following table to select a $GETKEYCH definition or function.

If you want to...	*then see...*	*page*
use the translated numeric keypad definitions	Cursor control keypad definitions.	5-14
obtain one translated character from the keyboard	$GETKEYCH.	5-15
turn shift inversion on or off	$GETKEYCHI.	5-16

Cursor control keypad definitions

The following definitions are the translated definitions of the numeric keypad into the cursor control keypad by $GETKEYCH.



Two other definitions are also provided. They are

- ***numberPad***, which is the first value in the cursor control keypad definitions, and
- ***afterNumberPad***, which is the value after the cursor control keypad definitions.

Example: The following example shows how these definitions can be used in a program segment.

```
IF numberPad <= char & char < afterNumberPad THEN
   CASE char {
      arrowUp    : moveUp();
      arrowLeft  : moveLeft();
      arrowRight : moveRight();
      arrowDown  : moveDown();
      }
 ELSE notCursorControl();
```

$GETKEYCH function

$GETKEYCH obtains one translated character from the keyboard. The character is not echoed on the screen.

Function Syntax	*Input*	*Output*
$GETKEYCH (h BYTE,	Horizontal position to obtain the character. Note: Valid inputs are 0 to 79.	
v BYTE	Vertical position to obtain the character. Note: Valid inputs are 0 to 23 on a 24 line screen.	
) CHAR;		Translated character.

Example:

```
function () :=
VAR   char CHAR;
{
   char := $GETKEYCH(15, 20);
```

$GETKEYCHI function

$GETKEYCHI turns shift inversion on or off.

Function Syntax	*Input*	*Output*
$GETKEYCHI (invert BOOLEAN);	TRUE turns shift inversion on, FALSE turns shift inversion off.	

Example:

```
function () :=
{
   $GETKEYCHI(TRUE);
```

$KEYCH

Introduction

The $KEYCH CUF is a general keyboard I/O package that contains functions to

- obtain one untranslated character from the keyboard,
- restore the keyboard translate pointer,
- allow or inhibit control of the cursor, and
- set a keyin timeout value.

$KEYCH functions

Use the following table to select a $KEYCH function.

If you want to...	*then see...*	*page*
obtain one untranslated character from the keyboard	$KEYCH.	5-18
restore the keyboard translation table pointer	$KEYCHR.	5-19
allow or inhibit control of the cursor	$KEYCHE.	5-20
set keyin timeout value	$KEYCHT.	5-21

$KEYCH function

$KEYCH obtains one untranslated character from the keyboard. The character is not echoed on the screen.

Note: This function changes the keyboard translation table pointer. Call $KEYCHR to restore this pointer.

Function Syntax	*Input*	*Output*
$KEYCH (		
horz BYTE,	Horizontal position to obtain the character. Note: Valid inputs are 0 to 79.	
vert BYTE	Vertical position to obtain the character. Note: Valid inputs are -12 to 11 on a 24 line screen.	
char ^ CHAR		Stores the untranslated character in char^. $WSTIMEO if time out occurred.
) D$CCODE;		$WSIO error if one occurs.

Example:

```
function () :=
VAR   char CHAR;
{
   IF $KEYCH(0, -12, &char) && D$CFLAG THEN $ERMSG();
```

$KEYCHR function

$KEYCHR restores the keyboard translation table pointer to the value it had before $KEYCH was called the first time. This function must be called before calling any keyin routines that expect translated characters. Example: $WSIO, $GETKEYCH, etc...

Function Syntax	*Input*	*Output*
$KEYCHR ( ) D$CCODE;		$WSIO error if one occurs.

Example:

```
function () :=
VAR   char CHAR;
{
   IF $KEYCH(0, -12, &char) && D$CFLAG THEN $ERMSG();
   IF $KEYCHR() && D$CFLAG THEN $ERMSG();
```

$KEYCHE function

$KEYCHE allows or inhibits $KEYCH from controlling the cursor.

Function Syntax	*Input*	*Output*

$KEYCHE (
  cursOn BOOLEAN
 	
  );
	TRUE enables $KEYCH to turn the cursor on and off, FALSE inhibits $KEYCH from controlling the cursor.	

Example:

```
function () :=
{
   $KEYCHE(TRUE);
```

$KEYCHT function

$KEYCHT sets the keyin timeout value in seconds.

Note: The keyin timeout value will default to $FOREVER if this function is not called.

Function Syntax	*Input*	*Output*
$KEYCHT (timeOut BYTE);	Number of seconds to wait for keyin character. Note: $FOREVER means not to time out.	

Example:

```
function () :=
{
   $KEYCHT(10);
```

$SEQFH

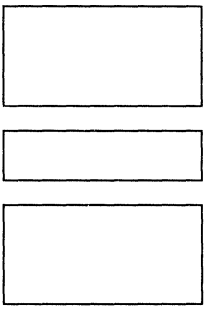
Introduction

The $SEQFH CUF is a sequential text file handling package that allows reading a text file forwards and backwards while inserting and deleting lines.

Input file

The sequential file handler manages the text file through three individual windows:

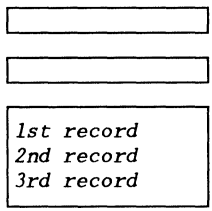
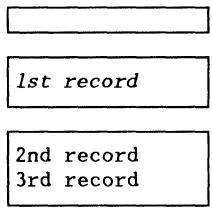
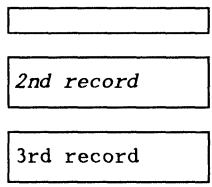
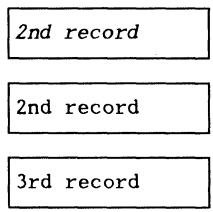
- The records *above* the current sequential record,
- the *current* sequential record, and
- the records *below* the current sequential record.



Your view of the input file is through the current sequential record window.

How the windows work

This diagram illustrates the three window file management:

Input File State	*Input File Diagram*
Initially, a file only has records in the below window.	
When you "get" a record from the below window, it becomes the current sequential record. The current sequential record is written into data space that you specify. It is not actually part of the file until it is "put" back into either the above or below window.	
Another "get from below" overstores the record that was in the current sequential window with the next record from the below window. Note: Successive "gets", without matching "puts", will delete records from the file.	
A "put record above" copies the record in the current sequential window into the above window. Note: Successive "puts", without matching "gets", will insert records into the file.	

$SEQFH types and functions

Use the following table to select a $SEQFH type or function.

If you want to...	*then see...*	*page*
declare a sequential file	SFH$IPT.	5-25
initialize the sequential file handler	$sfhInit.	5-26
get a record from the above window	$sfhGetA.	5-27
write the character string into the above window	$sfhPutA.	5-29
get a record from the below window	$sfhGetB.	5-30
write the character string into the below window	$sfhPutB.	5-32
force the records in the above and below windows to be written to the output file	$sfhQuit.	5-33
release all memory sectors used by the sequential file handler	$sfhRmem.	5-34

SFH$IPT type

SFH$IPT is defined as follows:

Type Definition	*Description*
TYPDEF SFH$IPT STRUCT { SFH$IMNS BYTE;	Maximum number of sectors to allocate in 4K quantities.
SFH$IEFP LONG;	Input end-of-file pointer.
SFH$IIN ^ $PFDB;	Address of the open input file $PFDB.
SFH$IOUT ^ $PFDB;	Address of the open output file $PFDB.
SFH$ISCR ^ $PFDB;	Address of the open scratch file $PFDB.
SFH$IASM BYTE;	Mapped sector number, MSN, of a 4K sector of memory to be used for buffering the portion of the input file above the current sequential record.
SFH$IBSM BYTE;	Mapped sector number, MSN, of a 4K sector of memory to be used for buffering the portion of the input file below the current sequential record.
SFH$IGET ^ D$CALLF;	Address of a memory allocation routine. Example: MEMGET$
SFH$IREL ^ D$CALLF; };	Address of a memory deallocation routine. Example: MEMREL$

$sfhInit function

$sfhInit initializes the sequential file handler.

Function Syntax	*Input*	*Output*
$sfhInit (sfhPtr ^ SFH$IPT	Address of an initialized sequential file descriptor.	
) D$CCODE;		D$CFLAG if there is insufficient memory available.

Example:

```
function () :=
VAR   sfhIpt SFH$IPT;
{
   sfhIpt.SFH$IMNS := 64;              /* 256 K max */
   sfhIpt.SFH$IEFP := D$GET24(&inOpenPt.$OTFLEN) << 8
                      || inOpenPt.$OTFEOFB;
   sfhIpt.SFH$IIN  := &inPfdb;
   sfhIpt.SFH$IOUT := &outPfdb;
   sfhIpt.SFH$ISCR := &scratchPfdb;
   sfhIpt.SFH$IASM := 016<<4;          /* 016-0167777 */
   sfhIpt.SFH$IBSM := 017<<4;          /* 017-0177777 */
   sfhIpt.SFH$IGET := &MEMGET$;
   sfhIpt.SFH$IREL := &MEMREL$;
   IF $sfhInit(&sfhIpt) && D$CFLAG THEN
      error('Insufficient memory available');
```

$sfhGetA function

$sfhGetA gets a record from the above window into the current sequential record buffer.

Function Syntax	*Input*	*Output*		
$sfhGetA (destPtr ^^ CHAR,	Address of a pointer to the last position in the current sequential record buffer. Note: This buffer must include space for two $LEORs.	Pointer destPtr^ points to the byte before the first byte stored in the current sequential record buffer.		
len BYTE	Size of the buffer.			
) D$CCODE;		*D$CFLAG*	*D$ZFLAG*	*Function Result*
		FALSE	——	No error.
		TRUE	FALSE	Maximum length overflow.
			TRUE	Beginning of file. One $LEOR obtained.

$sfhGetA function (continued)

Example: In the following program segment, $sfhGetA gets (or pops) the last record from the above window and stores it into the buffer.

```
function () :=
VAR   DEFINE(bufSize, 250)
      buffer [bufSize] CHAR;
      ptr ^ CHAR;
      flag D$CCODE;
{
   ptr := &buffer[bufSize-1];
   flag := $sfhGetA(&ptr, bufSize);
   IF flag && D$CFLAG THEN {
      IF flag && D$ZFLAG THEN display('Top of file.')
       ELSE error('Input record too large');
      }
    ELSE {
      n$write(n$WSOut, ptr+1, &buffer[bufSize]-ptr);
```

$sfhPutA function

$sfhPutA writes the character string into the above window.

Function Syntax	*Input*	*Output*
$sfhPutA (source ^ CHAR);	Address of the first character in the line storage buffer terminated by a $LEOR.	

Example: In the following program segment, $sfhPutA puts (or pushes) the record in the buffer into the above window.

```
function () :=
VAR   DEFINE(bufSize, 250)
      buffer [bufSize] CHAR;
      ptr ^ CHAR;
      n BYTE;
{
   ptr := &buffer[0];
   n := n$read(n$WSIn, ptr, bufSize);
   buffer[n-1] := $LEOR;
   $sfhPutA(ptr);
```

$sfhGetB function

$sfhGetB gets a record from the below window.

Function Syntax	*Input*	*Output*		
$sfhGetB (destPtr ^^ CHAR,	Address of a pointer to the first position in the current sequential record buffer. Note: This buffer must include space for one $LEOR.	Pointer destPtr^ points to the byte after the last byte stored in the current sequential record buffer.		
len BYTE	Size of the buffer.			
) D$CCODE;		*D$CFLAG*	*D$ZFLAG*	*Function Result*
		FALSE	—	No error.
		TRUE	FALSE	Maximum length overflow.
			TRUE	End of file. One $LEOR obtained.

$sfhGetB function (continued)

Example: In the following program segment, $sfhGetB gets (or pops) the first record from the below window and stores it into the buffer.

```
function () :=
VAR   DEFINE(bufSize, 250)
      buffer [bufSize] CHAR;
      ptr ^ CHAR;
      flag D$CCODE;
{
   ptr := &buffer[0];
   flag := $sfhGetB(&ptr, bufSize);
   IF flag && D$CFLAG THEN {
      IF flag && D$ZFLAG THEN display('End of file.')
       ELSE error('Input record too large');
      }
    ELSE {
      n$write(n$WSOut, &buffer[0], ptr-buffer[0]);
```

$sfhPutB function

$sfhPutB writes the character string into the below window.

Function Syntax	*Input*	*Output*
$sfhPutB (source ^ CHAR,	Address of the $LEOR following the last character in the line storage buffer.	
len BYTE);	Number of characters from the line storage buffer to write. Note: Including space for the $LEOR.	

Example: In the following program segment, $sfhPutB puts (or pushes) the record in the buffer into the below window.

```
function () :=
VAR   DEFINE(bufSize, 250)
      buffer [bufSize] CHAR;
      n BYTE;
{
   n := n$read(n$WSIn, &buffer[0], bufSize);
   buffer[n-1] := $LEOR;
   $sfhPutB(&buffer[n-1], n);
```

$sfhQuit function

$sfhQuit forces the records in the above and below windows to be written to the output file and puts an EOF mark into the output file.

Function Syntax	*Input*	*Output*

`$sfhQuit (`
`  eofPtr ^ LONG`
 	
`  );`
	`Address of a LONG variable.`	`Stores the output file EOF pointer in eofPtr^.`

Example:

```
function () :=
VAR   eof LONG;
{
   $sfhQuit(&eof);
```

$sfhRmem function

$sfhRmem releases all memory sectors used by the sequential file handling routines.

Function Syntax	*Input*	*Output*
`$sfhRmem (` `  );`		

Example:

```
function () :=
VAR   eof LONG;
{
   $sfhQuit(&eof);
   $sfhRmem();
```

Chapter 6.
MESSAGE HANDLING CUFS
OVERVIEW

Introduction

This chapter describes the message handling CUFs.

The message handling CUFs are used to handle message I/O.

Selecting a CUF

Use the following table to determine which message handling CUF to use.

If you want to...	*then use...*	*page*
• open a specific message file, • access specific messages, or • access messages sequentially	$MSG.	6-2
access specific message members from D$IO D$WRITE statements	$MSGIO.	6-6

$MSG

Introduction

The $MSG CUF is a message I/O package that can

- open a command file and search for a message member,
- copy a message from the message member into memory, and
- get sequential messages from the message member.

$MSG functions

Use the following table to select a $MSG function.

If you want to...	*then see...*	*page*
open a message member	$MSGOPEN.	6-3
get a specific message	$MSGGET.	6-4
get the next message	$MSGNEXT.	6-5

$MSGOPEN function

$MSGOPEN looks up a message member in the specified file.

Function Syntax	*Input*	*Output*
$MSGOPEN (pfdbPtr ^ $PFDB,	Address of a command file $PFDB. Note: If the $PFDB $FAVID is a $NOADR, it will open the executing command file.	
memNamePtr ^ $LNAMET	Address of the name of the message member.	
) D$CCODE;		D$CFLAG if I/O error.

Example:

```
function () :=
VAR   bufDesc $BufDesc;              /* $BUFFER CUF */
      STATIC mName $LNAMET := 'MESSAGE ';
{
   IF $alocBuf(&bufDesc, 256) THEN {
      bufDesc.pfdbP^.$PFVID := $PCRFVUP;
      IF $MSGOPEN(bufDesc.pfdbP, &mName)
        && D$CFLAG THEN $ERMSG();
      }
    ELSE error('Insufficient memory');
```

$MSGGET function

$MSGGET copies a specified message from the message member into memory.

Function Syntax	*Input*	*Output*
$MSGGET (msgNr UNSIGNED,	Number of the message to get.	
msgPtr ^ CHAR,	Address of the first character in the area to store the message.	Points to the message terminated by a $ES.
msgMaxLen BYTE	Maximum space available to store the message.	
) D$CCODE;		D$ZFLAG if message not found. D$CFLAG if I/O error.

Example:

```
function () :=
VAR   flag D$CCODE;
      message [80] CHAR;
{
   flag := $MSGGET(10, &message[0], SIZEOF message);
   IF flag && D$CFLAG THEN $ERMSG();
   IF flag && D$ZFLAG THEN error('Message not found');
```

$MSGNEXT function

$MSGNEXT copies the next sequential message from the message member into memory.

Function Syntax	*Input*	*Output*
$MSGNEXT (msgPtr ^ CHAR,	Address of the first character in the area to store the message.	Points to the message terminated by a $ES.
msgMaxLen BYTE	Maximum space available to store the message.	
) D$CCODE;		D$CFLAG if I/O error.

Example:

```
function () :=
VAR   i, n BYTE;
      message [80] CHAR;
{
   i := 0;
   LOOP {
      IF $MSGNEXT(&message[0], SIZEOF message)
        && D$CFLAG THEN $ERMSG();
      n := $SCAN(&message[0], $ES, SIZEOF message);
      n$format(n$WSOut, S(message, n), LN);
    WHILE ++i < 10;
      };
```

$MSGIO

Introduction

The $MSGIO CUF is used to access a message member through D$IO.

Open the message member first

The $MSGIO CUF uses the $MSG CUF to access the message member. You must open the message member with $MSGOPEN before using the $MSGIO CUF.

The "M" phrase type

$MSGIO adds the "M" or message phrase type to the D$IO D$WRITE phrases.

The "M" instructs the D$IO driver that a message number will follow as the next parameter. This message number is used to look up the corresponding message in the message member. The corresponding message is written to the output device.

Syntax for the "M" phrase is

M, <message number>

The "M" phrase type (continued)

Example:

```
pfdb $PFDB := { $NOADR, { 0, 0 }, 1, 0, 1,
   { { $NOPSK, (<UNSIGNED>&D$BUF1) >>8 } }
   };

function () :=
VAR   STATIC mName $LNAMET := 'MESSAGE ';
{
   IF $MSGOPEN(&pfdb, &mName) && D$CFLAG THEN $ERMSG();
   D$WRITE(&D$DSP, S, 'Message # 24 is ', M, 24, LN);
```

Chapter 7.
LIST PROCESSING CUFS
OVERVIEW

Introduction

This chapter describes the list processing CUFs.

The list processing CUFs are used to process linked lists and character queues.

Selecting a CUF

Use the following table to determine which list processing CUF to use.

If you want to...	*then use...*	*page*
manipulate data with doubly linked lists	$LINKS.	7-2
buffer characters on a queue	$QUEUE.	7-6

$LINKS

Introduction

The $LINKS CUF is a doubly linked list manipulation package.

$LINKS types and functions

Use the following table to select a $LINKS type or function.

If you want to...	*then see...*	*page*
declare a linked list descriptor	$ListHeader.	7-3
insert a node into the linked list	$LLINS.	7-4
delete a node from the linked list	$LLDEL.	7-5

$ListHeader type

$ListHeader is defined as follows:

Type Definition	*Description*
TYPDEF $ListHeader STRUCT { f ^ $ListHeader;	Forward pointer to the next linked list node.
r ^ $ListHeader; };	Reverse pointer to the previous linked list node.

Example: Linked lists are not useful unless they contain more data than the links. This example uses the linked list type as a field in a larger structure.

```
TYPDEF NameList STRUCT {
   links $ListHeader;
   name [30] CHAR;
   address [50] CHAR;
   id UNSIGNED;
   };
```

$LLINS function

$LLINS inserts a new node onto the linked list.

Function Syntax	*Input*	*Output*
$LLINS (new ^ $ListHeader,	Address of the new node to insert onto the linked list.	
after ^ $ListHeader);	Address of the node in the linked list to insert the new node after.	

Example:

```
start NameList;

function () :=
VAR   node NameList;
{
   $LLINS(&node.links, &start.links);
```

$LLDEL function

$LLDEL deletes a node from the linked list.

Function Syntax	*Input*	*Output*
$LLDEL (old ∧ $ListHeader	Address of the node to remove from the linked list.	
) ∧ $ListHeader;		Address of the node removed from the linked list.

Example:

```
function (pNode ∧ $ListHeader) :=
{
   $LLDEL(pNode);
```

$QUEUE

Introduction

The $QUEUE CUF is a set of functions for handling a circular character queue buffer.

$QUEUE types and functions

Use the following table to select a $QUEUE type or function.

If you want to...	*then see...*	*page*
declare a character queue descriptor	$Queue.	7-7
initialize a queue descriptor	$initQueue.	7-8
add a character to the queue	$addQueue.	7-9
delete a character from the queue	$remQueue.	7-10
determine the number of characters in the queue	$sizeQueue.	7-11
determine the number of character places left in the queue	$slackInQueue.	7-12
determine if the queue is full	$isQueueFull.	7-13
determine if the queue is empty	$isQueueEmpty.	7-14

$Queue type

$Queue is defined as follows:

Type Definition	*Description*
TYPDEF $Queue STRUCT { start ^ CHAR;	Pointer to the starting boundary of the queue.
end ^ CHAR;	Pointer to the ending boundary of the queue.
first ^ CHAR;	Pointer to the first character in the active queue.
after ^ CHAR;	Pointer to the character after the last character in the active queue.
};	

$initQueue function

$initQueue initializes a queue descriptor.

Function Syntax	*Input*	*Output*
$initQueue (q ∧ Queue,	Address of a queue descriptor.	Initialized queue descriptor with starting and ending boundaries, and active queue pointers.
buff ∧ CHAR,	Address of the first character in the character buffer to be used as the queue. Note: The buffer should be one byte larger than what you are intending to use.	
len UNSIGNED);	Number of characters in the buffer to be used as the queue.	

Example:

```
function () :=
VAR   queue $Queue;
      buffer [1000] CHAR;
{
   $initQueue(&queue, &buffer[0], SIZEOF buffer);
```

$addQueue function

$addQueue adds a character into the queue.

Function Syntax	*Input*	*Output*
$addQueue (q ^ $Queue,	Address of the queue descriptor.	
c CHAR	Character to be added to the queue.	
) BOOLEAN;		TRUE if successful in adding the character to the queue. FALSE otherwise.

Example:

```
function (pQueue ^ $Queue) :=
VAR   letter CHAR;
{
   letter := 'A';
   LOOP {
      IF ~ $addQueue(pQueue, letter) THEN
         error('No more room in the queue');
    WHILE letter++ < 'Z';
      };
```

$remQueue function

$remQueue removes the first character from the queue.

Function Syntax	*Input*	*Output*
$remQueue (q ^ $Queue,	Address of the queue descriptor.	
cPtr ^ CHAR	Address of a character.	Stores the character removed from the queue into cPtr^.
) BOOLEAN;		TRUE if successful in removing the character from the queue. FALSE otherwise.

Example:

```
function (pQueue ^ $Queue) :=
VAR   ch CHAR;
{
   IF $remQueue(pQueue, &ch) THEN
      n$format(n$WSOut, 'Removed ', C(ch), LN);
```

$sizeQueue function

$sizeQueue determines the number of characters in the queue.

Function Syntax	*Input*	*Output*
$sizeQueue (q ∧ $Queue	Address of the queue descriptor.	
) UNSIGNED;		Number of characters in the queue.

Example:

```
function (pQueue ∧ $Queue) :=
{
   n$format(n$WSOut, D( $sizeQueue(pQueue) ),
     ' characters in the queue', LN);
```

$slackInQueue function

$slackInQueue determines the number of available character places left in the queue.

Function Syntax	*Input*	*Output*
`$slackInQueue (` `  q ∧ $Queue`	`Address of the queue descriptor.`	
`  ) UNSIGNED;`		`Number of available character places left in the queue.`

Example:

```
function (pQueue ∧ $Queue) :=
{
   n$format(n$WSOut, D( $slackInQueue(pQueue) ),
     ' available characters in the queue', LN);
```

$isQueueFull function

$isQueueFull determines whether the queue buffer space has been exhausted.

Function Syntax	*Input*	*Output*
$isQueueFull (q ∧ $Queue	Address of the queue descriptor.	
) BOOLEAN;		TRUE if the queue buffer is full. FALSE otherwise.

Example:

```
function (pQueue ∧ $Queue) :=
{
   IF $isQueueFull(pQueue) THEN
      display('Queue is full');
```

$isQueueEmpty function

$isQueueEmpty determines whether the queue is empty.

Function Syntax	*Input*	*Output*
$isQueueEmpty (q ∧ $Queue;	Address of the queue descriptor.	
) BOOLEAN;		TRUE if the queue does not contain any entries. FALSE otherwise.

Example:

```
function (pQueue ∧ $Queue) :=
{
   IF $isQueueEmpty(pQueue) THEN
      display('Queue is empty');
```

Chapter 8.
PROGRAM STRUCTURING CUFS
OVERVIEW

Introduction

This chapter describes the program structuring CUFs.

The program structuring CUFs are used to conditionally compile code, simulate bit fields, etc...

Selecting a CUF

Use the following table to determine which message handling CUF to use.

If you want to...	*then use...*	page
exit your program with an internal error message written with D$IO if certain conditions in your program are not TRUE	$ASSERT.	8-2
exit your program with an internal error message written with NOSL if certain conditions in your program are not TRUE	$assert.	8-3
jump between two functions	$CATCH.	8-4
simulate bit fields	$FIELD.	8-8
simulate C language "for" loops	$FOR.	8-15
selectively compile parts of your program	IFINC.	8-19

$ASSERT

Introduction

The $ASSERT CUF is a single function which checks conditions in a program that should be TRUE. If a "FALSE assertion" occurs, $ASSERT will terminate the program with an internal error message through D$IO.

$ASSERT function

$ASSERT displays an internal error message and exits through $ERROR on a false boolean input.

Function Syntax	*Input*	*Output*
$ASSERT (flag BOOLEAN,	Boolean condition to examine.	
msg [any size] CHAR); 	Message to display before terminating program execution.	

Example:

```
function () :=
{
   $ASSERT(ptr ~= NIL, 'Pointer cannot be NIL');
```

$assert

Introduction

The $assert CUF is a single function which checks conditions in a program that should be TRUE. If a "FALSE assertion" occurs, $assert will terminate the program with an internal error message through the NOSL I/O package.

$assert function

$assert displays an internal error message and exits through $ERROR on a false boolean input.

Function Syntax	*Input*	*Output*
$assert (flag BOOLEAN,	Boolean condition to examine.	
msg [any size] CHAR); 	Message to display before terminating program execution.	

Example:

```
function () :=
{
   $assert(ptr ~= NIL, 'Pointer cannot be NIL');
```

$CATCH

Introduction

The $CATCH CUF is a machine dependent implementation of nonlocal goto's.

Note: If you need to use this CUF, you may be doing something wrong.

$CATCH types and functions

Use the following table to select a $CATCH type or function.

If you want to...	*then see...*	*page*
declare a goto location descriptor	$SAVE.	8-5
set a return address for a nonlocal goto	$CATCH.	8-5
jump to the address in the location descriptor	$THROW.	8-6

$SAVE type

The $SAVE type is nonportable and its contents need not be examined by your program.

$SAVE is defined as follows:

Type Definition	*Description*
TYPDEF $SAVE STRUCT { retPC ^ BYTE;	Return address to exit to.
prevSP ^ BYTE; };	Saved stack pointer.

$CATCH function

$CATCH sets a return address for a nonlocal goto.

Function Syntax	*Input*	*Output*
$CATCH (p ^ $SAVE	Address of an address and state descriptor.	Initialized address and state descriptor with current position info.
) BOOLEAN;		Always returns TRUE. Note: If $THROW is called with this address descriptor, program control returns to this $CATCH call with a FALSE result.

$THROW function

$THROW jumps to the specified address.

Function Syntax	*Input*	*Output*
$THROW (p ^ $SAVE	Address of the address and state descriptor to goto.	
) BOOLEAN;		Always FALSE. Note: This value actually gets set into $CATCH so that FALSE is returned by $CATCH after a $THROW.

Example of the $CATCH CUF

In the following example, $CATCH [1] is executed. It will set throwAddr [2] with the return address, and return a TRUE boolean condition. This causes function() to be called. If $THROW [3] is called, it will return through the $CATCH [1] call, returning a FALSE boolean condition which will result in $ERROR() [4] being called.

```
throwAddr $SAVE;                                          [2]

function () :=
VAR    s [80] CHAR;
       n BYTE;
{
   n := n$read(n$WSIn, &s[0], SIZEOF s);
   IF s[0] = ' ' THEN $THROW(&throwAddr);                 [3]
   :
   : (rest of the function)
   };

ENTRY MAIN () :=
{
   IF $CATCH(&throwAddr) THEN function()                  [1]
    ELSE $ERROR();                                        [4]
   };
```

$FIELD

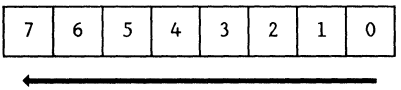
Introduction

The $FIELD CUF is a bit field simulation package which enables you to set up a named mask to define a bit or a set of adjacent bits in a single scalar to be a separate field.

Bit field numbering

The bit positions in a scalar are numbered starting at zero from least to most significant.

The following diagram shows the bit numbers of a byte.



$FIELD macros and functions

Use the following table to select a $FIELD macro or function.

If you want to...	*then see...*	*page*
define a field identifier	field.	8-9
get a field	fieldGet.	8-10
test a field for nonzeros	fieldTest.	8-11
store a value into a field	fieldStore.	8-12
set a field	fieldSet.	8-13
clear a field	fieldClear.	8-14

field macro

field defines an identifier to be a location and width of a field within a scalar.

Function Syntax	*Input*	*Output*
field (name Identifier,	A DASL identifier. Note: Be careful not to use the same name more than once.	
offset BYTE,	Bit position of the field.	
size BYTE)	Number of bits in the field.	

Example: The following program segment defines six different bit fields bit0, bit1, ..., bits5to7. All but the last field are one bit masks. bits5to7 is a three bit mask.

```
function () :=
VAR   byte BYTE;
         field(bit0, 0, 1)
         field(bit1, 1, 1)
         field(bit2, 2, 1)
         field(bit3, 3, 1)
         field(bit4, 4, 1)
         field(bits5to7, 5, 3)
{
```

fieldGet function

fieldGet gets the field defined by the identifier from a DASL scalar.

Function Syntax	*Input*	*Output*
fieldGet (value Scalar,	DASL scalar that contains the field.	
name Identifier	Name of the mask that identifies the field within the scalar.	
) Scalar;		Contents of the field.

Example: The following program segment gets the high order 8 bit field of the integer "i".

```
field(highByte, 8, 8)

function (i INT) :=
VAR   fld BYTE;
{
   fld := fieldGet(i, highByte);
```

fieldTest function

fieldTest tests for nonzeros in a field.

Function Syntax	*Input*	*Output*
fieldTest (value Scalar,	DASL scalar that contains the field to evaluate.	
name Identifier	Name of the mask that identifies the field within the scalar.	
) BOOLEAN;		TRUE if the value of the field is nonzero. FALSE otherwise.

Example:

```
field(prtFlag, 13, 1)

function (flags UNSIGNED) :=
{
   IF fieldTest(flags, prtFlag) THEN printIt();
```

fieldStore function

fieldStore stores a value into a field.

Function Syntax	*Input*	*Output*
fieldStore (valuePtr ^ Scalar,	Address of the DASL scalar that contains the field to be stored into.	
name Identifier,	Name of the mask that identifies the field within the scalar.	
value Scalar	Value to be stored into the field.	
) Scalar;		The input DASL scalar with the updated field.

Example:

```
field(highByte, 8, 8)

function (i UNSIGNED) :=
{
   fieldStore(&i, highByte, 0377);
```

fieldSet function

fieldSet sets the bits in the field of a scalar to ones.

Function Syntax	*Input*	*Output*
fieldSet (valuePtr ^ Scalar,	Address of the DASL scalar that contains the field to be set.	
name Identifier,	Name of the mask that identifies the field within the scalar.	
) Scalar;		The input DASL scalar with the updated field.

Example:

```
field(prtFlag, 13, 1)

function (flags UNSIGNED) :=
{
   fieldSet(&flags, prtFlag);
```

fieldClear function

fieldClear clears the bits in a field to zeros.

Function Syntax	*Input*	*Output*
fieldClear (valuePtr ^ Scalar,	Address of the DASL scalar that contains the field to be cleared.	
name Identifier,	Name of the mask that identifies the field within the scalar.	
) Scalar;		The input DASL scalar with the updated field.

Example:

```
field(prtFlag, 13, 1)

function (flags UNSIGNED) :=
{
   fieldClear(&flags, prtFlag);
```

$FOR

Introduction

The $FOR CUF is a set of macros which simulate the C language "for" statement.

$FOR macros

Use the following table to select a $FOR macro.

If you want to...	*then see...*	*page*
begin "for" loop structure	$for.	8-16
end "for" loop structure	$next.	8-16

$for macro

$for designates the beginning of the "for" loop structure. It is parameterized with

- a preloop initialization statement,
- a conditional exit expression, and
- an incrementation statement.

Macro Syntax	*Parameter Description*
$for (initialization statement,	Statement that will be executed prior to the loop structure.
termination expression,	Boolean condition expression that will terminate execution of the loop when it becomes FALSE.
incrementation statement);	Last statement in the loop to be executed.

$next macro

$next designates the end of the "for" loop structure.

Macro Syntax	*Parameter Description*
$next;	

Statements located between the $next and matching $for macros make up the loop body.

$FOR loop structure

A $FOR loop is opened with a $for and closed with a $next.

```
$for (statement, expression, statement);
    .
    .  (DASL code)
    .
$next;
```

Nested $FOR loops

$FOR loops may be nested within one another.

```
$for (statement, expression, statement);
    .
    .  (DASL code)
    .
    $for (statement, expression, statement);
        .
        .  (DASL code)
        .
    $next;
    .
    .  (DASL code)
    .
$next;
```

Cautions

Users of the $FOR loops should be aware that $for...$next loops need to be enclosed in braces if they are used where a single statement is required.

Example:

```
IF i = 0 THEN {
   $for (i := 0, i < 100, i++);
      n$format(n$WSOut, D(i), LN);
   $next;
   };
```

Expanded DASL code

This is an example of the $FOR and the resultant expanded DASL code.

```
function () :=
VAR   i BYTE;
{
   $for (i := 0, i < 100, i++);
      n$format(n$WSOut, D(i), LN);
   $next;
```

Which expands to...

```
function () :=
VAR   i BYTE;
{
   i := 0;
   LOOP {
    WHILE i < 100;
      n$format(n$WSOut, D(i), LN);
      i++;
      };
```

IFINC

Introduction

The IFINC CUF is a set of macros which enables selective compilation of DASL code.

IFINC definition file

The IFINC definition file has a "/TEXT" extension.

Example:

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(IFINC/TEXT)
:
: (rest of the program)
```

IFINC macros

Use the following table to select a IFINC macro.

If you want to...	*then see...*	*page*
begin a decision block	$if.	8-20
designate an alternate decision block branch	$else.	8-21
end a decision block	$endif.	8-21

$if macro

$if designates the beginning of the decision block for selective compilation.

Note: $if must be preceded by "$esc*/".

Macro Syntax	*Parameter Description*
$esc*/ $if (first characters to compare,	A sequence of characters or defined name of the first parameter to compare.
second characters to compare	A sequence of characters or defined name of the first parameter to compare.
)	

If the two parameters contain exactly the same characters, the code immediately following this macro will be compiled.

Note: Blank characters are significant.

$else macro

$else designates the alternate branch of the decision block.

Note: $else must be preceded by "$esc*/".

Macro Syntax	*Parameter Description*
`$esc*/ $else`	

The $else macro serves two purposes:

- it terminates the condition block that began at the matching $if macro, and
- it designates the beginning of another condition block which ends at the matching $endif macro.

$endif macro

$endif designates the end of the decision block.

Note: $endif must be preceded by "$esc*/".

Macro Syntax	*Parameter Description*
`$esc*/ $endif`	

Statements located between the $endif and matching $if macros make up the decision block body.

Decision block structure

A decision block is opened with a $if and closed with a $endif.

```
$esc*/ $if (parameter1, parameter2)
    .
    . (DASL code)
    .
$esc*/ $endif
```

The code inside the decision block is compiled if the $if parameters are equal.

Alternate decision branch

A decision block may contain an alternate decision branch, $else.

```
$esc*/ $if (parameter1, parameter2)
    .
    .  (DASL code)
    .
$esc*/ $else
    .
    .  (DASL code)
    .
$esc*/ $endif
```

The code between the $if and the $else is compiled if the $if parameters are equal. Otherwise, the code between the $else and the $endif is compiled.

Nested decision blocks

Decision blocks may be nested within one another.

```
$esc*/ $if (parameter1, parameter2)
   .
   .  (DASL code)
   .
   $esc*/ $if (parameter1, parameter2)
      .
      .  (DASL code)
   .  .
   $esc*/ $endif
   .
   .  (DASL code)
   .
$esc*/ $endif
```

Example

This is an example of the IFINC CUF. The compiled lines are bold faced.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(IFINC/TEXT)

DEFINE(off,0)
DEFINE(on,1)
DEFINE(NOSL,on)          /* Set either "on" or "off" */
DEFINE(LOGGING,off)      /* Set either "on" or "off" */

$esc*/ $if(NOSL,on)
INCLUDE(N$/DEFS)
$esc*/ $else
INCLUDE(D$IO)
$esc*/ $endif

ENTRY MAIN () :=
{
   $esc*/ $if(NOSL,on)
      $esc*/ $if(LOGGING,on)
   n$format(n$LogOut, 'NOSL is on, LOGGING is on', LN);
      $esc*/ $else
   n$format(n$WSOut, 'NOSL is on, LOGGING is off', LN);
      $esc*/ $endif
   $esc*/ $else
      $esc*/ $if(LOGGING,on)
   D$WRITE(&D$OUT, 'NOSL is off, LOGGING is on', LN);
      $esc*/ $else
   D$WRITE(&D$DSP, 'NOSL is off, LOGGING is off', LN);
      $esc*/ $endif
   $esc*/ $endif
   };
```

Reserved words

In addition to $esc, $if, $else, and $endif, the following names are DEFINEd in the IFINC CUF and should not be used.

- $cat
- $push
- $pop
- $ifStk
- $cond
- $getCond
- $zap
- $doIf

Chapter 9.
SCHEDULER CUFS
OVERVIEW

Introduction

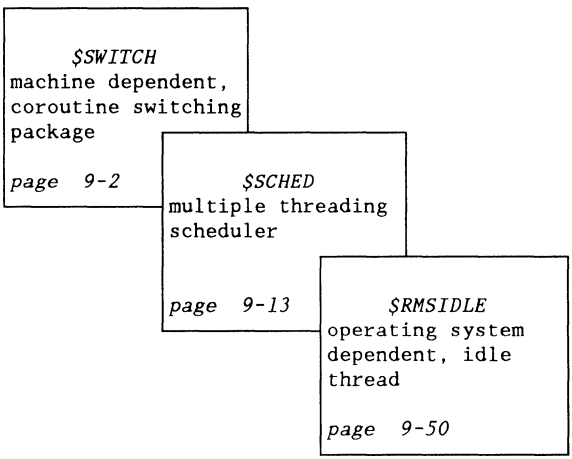
This chapter describes the scheduler CUFs.

The scheduler CUFs are used to schedule multiple programs running in a single RMS task.

Scheduler layers

The scheduler has been broken up into three CUFS in order to separate the actual scheduler from the machine dependent and operating system dependent functions.

The separate parts of the scheduler are layered as illustrated in the following diagram.



$SWITCH

Introduction

The $SWITCH CUF is the machine dependent, coroutine switching part of the scheduler.

What is a coroutine?

A coroutine, or concurrent routine, is a routine running simultaneously with one or more other routines.

The term "concurrent" is somewhat misleading since the coroutines do not actually run simultaneously. Each coroutine runs independently until an instruction is given to pass execution control to another coroutine. When execution control is returned to the originating coroutine, it resumes execution at the point it gave up control. All other coroutines are "asleep" until they are "awakened" by the one active coroutine.

The term "routine" simply means a sequence of instructions. The instructions can be part of one or more functions.

$SWITCH classifications, types, variables, and functions

Use the following table to select a $SWITCH classification, type, variable, or function.

If you want to...	*then see...*	*page*
declare a function REENTRANT	REENTRANT.	9-4
define the message parameter descriptor	u$Param.	9-4
declare a coroutine stack descriptor	$StackDesc.	9-5
use the current stack descriptor	$coStackDesc.	9-5
initialize the program to be a coroutine	$coInit.	9-6
create a new coroutine	$coFork.	9-7
switch to another coroutine	$coSwitch.	9-9
save the state of a coroutine	$coSave.	9-10
restore the state of a coroutine	$coRestore.	9-11

REENTRANT function classification

REENTRANT is a defined name that is available for you to use as a function classification.

Macro Definition	*Description*
DEFINE(REENTRANT,RECURSIVE)	Function classification for functions that initiate new coroutines and for functions that may be suspended in one coroutine while another coroutine is running.

Example:

```
REENTRANT function () :=
{
```

u$Param type

The u$Param type is a type that must be declared by you. It is used for passing messages or variables between coroutines.

Example: If you wanted to pass 100 byte arrays between coroutines, you would define u$Param like this:

```
TYPDEF u$Param [100] BYTE;
```

$StackDesc type

The $StackDesc type is needed to define variables used by the $SWITCH functions. The contents, however, do not need to be set or examined by your program.

$StackDesc is defined as follows:

Type Definition	*Description*
TYPDEF $StackDesc STRUCT { framePtr ^ $Frame;	Points to the function stack information. Note: $Frame type is defined in the $SWITCH text. You will not need to examine variables of this type so its description has not been made available.
stacklst ^ BYTE; };	Points to the first byte of the stack information.

$coStackDesc variable

$coStackDesc is declared as part of the $SWITCH CUF and is available for you to examine.

Variable Definition	*Description*
$coStackDesc ^ $StackDesc;	Pointer to the stack descriptor of the currently active coroutine.

$coInit function

$coInit initializes the running program to be a separate coroutine.

Note: This function must be called prior to all other calls to the coroutine functions. It should also be called prior to any calls to REENTRANT or RECURSIVE functions.

Function Syntax	*Input*	*Output*
$coInit (stackDesc ^ $StackDesc,	Address of a stack descriptor.	Initialized descriptor with information about the stack.
stack ^ BYTE,	Address of the first byte of the stack storage area.	
len UNSIGNED);	Length of the stack storage area.	

Example:

```
routine1 $StackDesc;
stack1 [500] BYTE;

function () :=
{
   $coInit(&routine1, &stack1[0], SIZEOF stack1);
```

$coFork function

$coFork creates a new coroutine.

- The newly created coroutine begins executing from the statement following the $coFork.
- $coFork must be called from within a REENTRANT function that is defined to return a pointer to a u$Param.
- The newly created coroutine must never return from the function which called $coFork or a fatal error will occur. Processing may be terminated by switching to another coroutine. See $coSwitch for further details.
- The coroutine that calls $coFork falls asleep. If execution control is passed back to this coroutine, it resumes execution as if it was returning from the function that called $coFork. It does not execute the remaining code in the function following the $coFork.

Function Syntax	*Input*	*Output*
$coFork (stackDesc ^ $StackDesc,	Address of a stack descriptor.	Initialized descriptor with information about the created coroutine.
stack ^ BYTE,	Address of the first byte of the stack storage area.	
len UNSIGNED);	Length of the stack storage area.	

$coFork function (continued)

Example: The following segment of code calls the $coFork function.

```
TYPDEF u$Param BYTE;

routine2 $StackDesc;
stack2 [500] BYTE;

REENTRANT function () ^ u$Param :=
{
   $coFork(&routine2, &stack2[0], SIZEOF stack2);
```

$coSwitch function

$coSwitch passes execution control from one coroutine to another. The input parameter, params, will be the RESULT of the function that wakes up in the resumed coroutine.

Function Syntax	*Input*	*Output*
$coSwitch (stackDesc ^ $StackDesc,	Address of the stack descriptor to switch to.	
params ^ u$Params	Address of a parameter you are passing into the coroutine you are switching to.	
) ^ u$Params;		Pointer to a parameter passed back to this coroutine when it resumes execution.

Example:

```
TYPDEF u$Param BYTE;

routine2 $StackDesc;
stack2 [500] BYTE;

REENTRANT function () ^ u$Param :=
{
   $coFork(&routine2, &stack2[0], SIZEOF stack2);
   function1();
   function2();
   $coSwitch(&routine1, NIL);
```

$coSave function

$coSave saves the stack information of the current coroutine.

$coSave and $coRestore are lower level functions used by $coSwitch. You will probably not need to call these functions directly.

Function Syntax	*Input*	*Output*
`$coSave ( );`		

Example:

```
function () :=
{
   $coSave();
```

$coRestore function

$coRestore restores the stack information which

- causes the executing coroutine to go to sleep, and
- causes the coroutine described by stackDesc to resume execution.

Note: $coRestore must be called from within a REENTRANT function. The restored coroutine does not actually resume execution until the REENTRANT function returns. The function may also specify a return parameter for passing information between the coroutines. RESULT should be assigned after the $coRestore.

$coSave and $coRestore are lower level functions used by $coSwitch. You will probably not need to call these functions directly.

Function Syntax	*Input*	*Output*
$coRestore (stackDesc ^ $StackDesc,);	Address of the stack descriptor of the coroutine to resume execution.	

Example:

```
routine2 $StackDesc;
stack2 [500] BYTE;

REENTRANT function () ^ u$Param :=
{
   $coSave();
   $coRestore(&routine2);
   RESULT := NIL;
   };
```

Example of the $SWITCH CUF

The following program segment demonstrates a two coroutine program. The first coroutine (bold faced lines) is created by the $coInit function call [1] and falls asleep after creating a second coroutine by $coFork [2]. The second coroutine (italicized lines) begins after the call to $coFork [3] and continues until the call to $coSwitch [4]. The $coSwitch returns execution control to the first coroutine which resumes as though it was returning from the function [5]. Note that the line [6] following the $coSwitch is never executed.

```
TYPDEF u$Param BYTE;

routine1 $StackDesc;
stack1 [500] BYTE;

routine2 $StackDesc;
stack2 [500] BYTE;

REENTRANT function () ^ u$Param :=
{
   function0();
   $coFork(&routine2, &stack2[0], SIZEOF stack2);  [2]
   function1();                                    [3]
   function2();
   $coSwitch(&routine1, NIL);                      [4]
   function3();                                    [6]
   };

ENTRY MAIN () :=
VAR   pB ^ u$Param;
{
   $coInit(&routine1, &stack1[0], SIZEOF stack1);  [1]
   pB := function();
   function4();                                    [5]
   function5();
   };
```

$SCHED

Introduction

The $SCHED CUF is a set of functions that enables multiple threads to coexist within one RMS task.

$SCHED definition file

The $SCHED definition file must be included after the $LINKS and $STRING definition files.

Note: The $SCHED definition file includes the $SWITCH definition file.

Example: Your DASL include directives might look like this:

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE($STRING/DEFS)
INCLUDE($LINKS/DEFS)
INCLUDE($SCHED/DEFS)
:
: (rest of the program)
```

Overview of the $SCHED documentation

This table contains an overview of the $SCHED documentation.

To learn more about...	*see page...*
scheduler terminology	9-15
scheduler properties	9-16
scheduler initialization and control	9-17
scheduler synchronization and message passing	9-30
scheduler tickling	9-40
scheduler logging	9-43
scheduler termination	9-49

Scheduler terminology

$SCHED contains terminology that you may not be familiar with. Here is a rundown of a few of the terms used:

- ***thread*** – A thread is a separate process running within a single RMS task. A thread can be in one of three states:
  - active (there is only one active thread at one time),
  - waiting to become active, or
  - waiting for a signal or a message from another thread.
- ***execution control*** – Execution control refers to the period of time when a thread is actually running as the RMS task.
- ***ready queue*** – The ready queue is a queue for threads which are ready to execute but are waiting to be scheduled.
- ***semaphore*** – A semaphore is a communications device that is used to synchronize one thread with another.
- ***message*** – A message is data that is transferred between threads and is synchronized by a semaphore.
- ***feather*** – A feather is a variable that can be used to transfer information to threads. Feathers are primarily useful for informing threads that they should abort processing or stop waiting for a signal or a message.
- ***tickle*** – Tickle refers to the sending of a feather to a thread.

Scheduler properties

$SCHED has several properties that will help you to understand this scheduler:

- ***run to completion*** – Each thread initiated in the scheduler executes to completion unless it gives up execution control. In other words, a thread runs without being interrupted by the scheduler. It is the responsibility of each thread to give up execution control to the scheduler if other threads are expected to execute.
- ***prioritized*** – Each thread has a priority that is specified at the time of creation. The scheduler schedules higher priority threads to execute before lower priority threads.
- ***first come, first serve*** – Within each priority, waiting threads are allowed to execute in a first come, first serve basis.

Scheduler initialization and control

This section describes the variables, types, and functions that are provided for scheduler initialization and control.

Use the following table to select a $SCHED constant, type, variable, or function.

If you want to...	*then see...*	*page*
determine the maximum priority	$maxPriority.	9-18
declare a priority for a thread	$Priority.	9-18
define the private data descriptor	u$SchedData.	9-19
declare a thread control block descriptor	$ThreadCtlBlk.	9-20
use the current thread control block	$curTCB.	9-21
declare a thread entry point	$Thread.	9-21
initialize the program to be a thread	$initSched.	9-22
start a new thread	$fork.	9-23
determine if there are threads waiting to be scheduled	$wouldPass.	9-25
pass execution control to any waiting threads	$pass.	9-26
define a function that will execute before a scheduled thread	u$threadIn.	9-27
define a function that will execute after a scheduled thread	u$threadOut.	9-28
release thread control block memory	u$relTcb.	9-29

$maxPriority constant

$maxPriority is defined as a constant and is available for you to use.

Constant Definition	*Description*
DEFINE($maxPriority, 7)	Maximum priority of a thread.

$Priority type

$Priority is defined as follows:

Type Definition	*Description*
TYPDEF $Priority BYTE;	The priority of the thread. It is used by the scheduler to determine the order to execute the threads. 0 is the lowest priority, $maxPriority is the highest.

u$SchedData type

The u$SchedData type is a type that must be defined by you. Variables of this type may be used to store information that is unique to individual threads.

Example: This example uses u$SchedData to keep information about NOSL streams which are unique to each thread. Any commonly used function that needs to do any NOSL I/O can use this information without knowing which thread it is processing.

```
TYPDEF u$SchedData STRUCT {
   stream n$Stream;
   ptr ^ BYTE;
   len BYTE;
   };
```

$ThreadCtlBlk type

The $ThreadCtlBlk type is needed to define variables used by the $SCHED functions. All fields, except privData, do not need to be set or examined by your program.

$ThreadCtlBlk is defined as follows:

Type Definition	*Description*
TYPDEF $ThreadCtlBlk STRUCT { link $ListHeader;	A link that is used to place this thread control block on a queue.
msg ^ $Message;	Points to any message that has been received.
privData ^ u$SchedData;	Points to data that is initialized and used by you.
pending ^ $Sema4;	Points to the semaphore this thread is waiting on. NIL if the thread is not waiting.
feather ^ u$Feather;	Points to the data passed into the thread if it has been tickled.
priority $Priority;	Priority of this thread.
stack $StackDesc;	Describes the stack for this thread.
stackData [$stackSize] BYTE;	Storage area for the recursive stack.
name $String; };	Name used for logging scheduler events.

$curTCB variable

$curTCB is declared as part of the $SCHED CUF and is available for you to examine.

Variable Definition	*Description*
`$curTCB ^ $ThreadCtlBlk;`	`Pointer to the currently active thread control block.`

$Thread type

Use the $Thread type to declare functions that are entry points of new threads.

$Thread is defined as follows:

Type Definition	*Description*

`TYPDEF $Thread (`
`  feather ^ u$Feather`
 	
`  );`
	`Address of a feather. Will indicate whether a tickle occurred sometime between when this function was placed on the ready queue and when it actually began executing. NIL if no tickle occurred.`

Example:

```
function $Thread :=
(
```

$initSched function

$initSched initializes the running program to be a scheduled thread. $curTCB is left pointing at the thread control block that describes this thread.

Note: This function must be called prior to all other calls to the scheduler. It should also be called prior to any calls to REENTRANT or RECURSIVE functions.

Function Syntax	*Input*	*Output*
$initSched (priority $Priority,	Priority of this thread. Note: Valid inputs are 0 to $maxPriority.	
name ^ $String	Address of the name for this thread which will be used for scheduler logging.	
);		

Example:

```
function () :=
VAR   STATIC name $String := $str('Main Thread');
{
   $initSched(1, &name);
```

$fork function

$fork starts up a new thread. The new thread is placed at the end of the ready queue where it begins execution once the current thread, threads with higher priorities, and threads with equal priority that were already on the ready queue yield execution control.

Function Syntax	*Input*	*Output*
$fork (fp ^ $Thread,	Address of the function to begin executing as the new thread.	
newTCB ^ $ThreadCtlBlk,	Address of a thread control block. Note: The privData field should be initialized.	Initialized thread control block with information about the new thread.
priority $Priority,	Priority of the this thread. Note: Valid inputs are 0 to $maxPriority.	
name ^ $String); 	Address of the name for this thread which will be used for scheduler logging.	

$fork function (continued)

Example:

```
TYPDEF u$SchedData STRUCT {
   stream n$Stream;
   ptr ^ BYTE;
   len BYTE;
   };

threadOne $Thread :=
{
   .
   .  (DASL code)
   .
   };

function () :=
VAR   tcbl $ThreadCtlBlk;
      privData u$SchedData;
      string [80] CHAR;
      STATIC name $String := $str('Thread One');
{
   tcbl.privData := &privData;
   tcbl.privData^.stream := n$WSOut;
   tcbl.privData^.ptr := string[0];
   tcbl.privData^.len := SIZEOF string;
   $fork(&threadOne, &tcbl, 1, &name);
```

$wouldPass function

$wouldPass determines if there are threads waiting to be scheduled at or above the same priority on the ready queue.

Function Syntax	*Input*	*Output*
$wouldPass ( ) BOOLEAN;		TRUE if there are threads ready to be scheduled. FALSE otherwise.

Example:

```
function () :=
{
   IF ~ $wouldPass() THEN
      display('No threads on the ready queue.');
```

$pass function

$pass causes the active thread to give up execution control to any threads on the ready queue with higher or equal priority. If no such threads exist, the current thread resumes execution control.

Function Syntax	*Input*	*Output*
$pass ( ) ^ u$Feather;		Returns the address of a feather if this thread was tickled while asleep. NIL if not tickled.

Example:

```
function () :=
{
   IF $pass() ~= NIL THEN
      display('I was tickled awake.')
    ELSE display('The scheduler awakened me.');
```

u$threadIn function

u$threadIn is a function that must be declared by you. It is called right after a thread is scheduled and may be written for any purpose.

Note: u$threadIn must be declared "ENTRY".

The scheduler expects a function with the following syntax:

Function Syntax	*Input*	*Output*
u$threadIn ( );		

Example: The following program segment is the minimum u$threadIn. This is a perfectly acceptable definition for this function.

```
ENTRY u$threadIn :=
{
   };
```

u$threadOut function

u$threadOut is a function that must be declared by you. It is called right before a thread is unscheduled and may be written for any purpose.

Note: u$threadOut must be declared "ENTRY".

The scheduler expects a function with the following syntax:

Function Syntax	*Input*	*Output*
u$threadOut ( );		

Example: The following program segment is the minimum u$threadOut. This is a perfectly acceptable definition for this function.

```
ENTRY u$threadOut :=
{
   };
```

u$relTcb function

u$relTcb is a function that must be declared by you. It should release the memory used by the thread control block which is pointed to by $curTCB.

Note: u$relTcb must be declared "ENTRY".

The scheduler expects a function with the following syntax:

Function Syntax	*Input*	*Output*
`u$relTcb (` `  );`		

Example: The following program segment uses the $BUDDY CUF to deallocate the memory used by the thread control block.

```
ENTRY u$relTcb :=
{
   $free($curTCB, SIZEOF $curTCB^);
   };
```

Scheduler synchronization and message passing

Semaphores and messages are used to synchronize and transfer information between threads.

Types and functions

This section describes the types and functions provided for scheduler synchronization and message passing.

Use the following table to select a $SCHED type or function.

If you want to...	*then see...*	*page*
declare a semaphore descriptor	$Sema4.	9-31
declare a message descriptor	$Message.	9-32
initialize a semaphore	$initSema4.	9-33
signal a semaphore	$signal.	9-34
wait for a semaphore	$waitOn.	9-35
send a message	$sendMsg.	9-37
wait for a message	$waitMsg.	9-38

$Sema4 type

The $Sema4 type is needed to define variables used by the $SCHED functions. The contents, however, do not need to be set or examined by your program.

$Sema4 is defined as follows:

Type Definition	*Description*
TYPDEF $Sema4 STRUCT { count INT;	A counter used to synchronize threads.
link $ListHeader;	A link used to queue threads that are waiting for this semaphore to be signaled.
name $String; };	A name that is used for logging scheduler events.

$Message type

$Message is defined as follows:

Type Definition	*Description*
TYPDEF $Message STRUCT { link $ListHeader;	A link that is used to save this message for a thread. Note: This field will be maintained by the scheduler.
name $String;	A name that will be used for logging scheduler events. Note: This field must be initialized by your program.
};	

In order to pass messages between threads, a field of type $Message should be declared as the first member in a structure which contains the elements of your message.

Example: If you need to pass an array of 10 characters between threads, your message type might look like this:

```
TYPDEF Message STRUCT {
   msgCtl $Message;
   msgStr [10] CHAR;
   };
```

$initSema4 function

$initSema4 initializes a semaphore so that it can be used by the scheduler.

Function Syntax	*Input*	*Output*
$initSema4 (ptr ∧ $Sema4,	Address of semaphore to initialize.	Initialized semaphore.
name ∧ $String	Address of a name for this semaphore which will be used for scheduler logging.	
);		

Example:

```
function () :=
VAR    sema4 $Sema4;
       STATIC name $String := $str('Semaphore 1');
{
   $initSema4(&sema4, &name);
```

$signal function

$signal uses a semaphore to let another thread know that a certain event has taken place. If a thread has been waiting for this signal, $signal places it onto the ready queue. Otherwise, the signal is saved so a thread will know this event has taken place.

Function Syntax	*Input*	*Output*
$signal (ptr ^ $Sema4);	Address of the semaphore to signal.	

Example:

```
sema4 $Sema4;

function () :=
{
   $signal(&sema4);
```

$waitOn function

$waitOn checks to see if a semaphore has been signaled to determine if a certain event has taken place in another thread. If the semaphore has been signaled, the thread continues to execute. Otherwise, the thread gives up execution control and waits for a signal.

Function Syntax	*Input*	*Output*
$waitOn (ptr ^ $Sema4,	Address of the semaphore to wait on.	
completed ^ BOOLEAN	Address of a boolean.	TRUE if the semaphore you were waiting on was signaled. FALSE if tickled before receiving the message. Note: This flag can be TRUE even if the thread was tickled.
) ^ u$Feather;		Returns the address of a feather if this thread was tickled while asleep. NIL if not tickled.

$waitOn function (continued)

Example:

```
sema4 $Sema4;

function () :=
VAR   completed BOOLEAN;
      feather ^ u$Feather;
{
   feather := $waitOn(&sema4, &completed);
   IF feather ~= NIL THEN
      display('I was tickled awake.');
   IF completed THEN
      display('I received the signal.')
    ELSE display('I did not receive the signal.');
   };
```

$sendMsg function

$sendMsg uses a semaphore to let another thread know that a message has been sent to it. If a thread has been waiting for this message, $sendMsg places the thread onto the ready queue. Otherwise, the message is saved so a thread will know this message has been sent.

Function Syntax	*Input*	*Output*
$sendMsg (ptr ^ $Sema4,	Address of the semaphore to signal.	
msg ^ $Message);	Address of the message to send.	

Example:

```
TYPDEF Message STRUCT {
   msgCtl $Message;
   msgStr [10] CHAR;
   };
message Message;

sema4 $Sema4;

function () :=
VAR   STATIC name $String := $str('Message 1');
{
   n$read(n$WSIn, &message.msgStr[0], 10);
   message.msgCtl.name := name;
   $sendMsg(&sema4, &message.msgCtl);
```

$waitMsg function

$waitMsg checks to see if a semaphore has been signaled to determine if a certain message has been sent by another thread. If the semaphore has been signaled, the thread receives the message and continues to execute. Otherwise, the thread gives up execution control and waits for the message.

Function Syntax	*Input*	*Output*
$waitMsg (ptr ^ $Sema4,	Address of the semaphore to wait on.	
msgPtr ^^ $Message	Address of a pointer to a message.	Points to the message sent by the sending thread. NIL if tickled before receiving the message. Note: This pointer can point to a message even if the thread was tickled.
) ^ u$Feather;		Returns the address of a feather if this thread was tickled while asleep. NIL if not tickled.

$waitMsg function (continued)

Example:

```
TYPDEF Message STRUCT {
   msgCtl $Message;
   msgPtr ^ [80] CHAR;
   };
message Message;

sema4 $Sema4;

function () :=
VAR mPtr ^ $Message;
    feather ^ u$Feather;
{
   feather := $waitMsg(&sema4, &mPtr);
   IF feather ~= NIL THEN
      display('I was tickled awake.');
   IF mPtr ~= NIL THEN
      display((<^Message>mPtr)^.msgStr)
    ELSE display('I did not receive the message.');
```

Scheduler tickling

Tickling refers to the sending of a feather to a thread. Tickling is primarily useful for informing threads that they should abort processing or stop waiting for a signal or a message.

Types and functions

This section describes the types and functions provided for scheduler tickling.

Use the following table to select a $SCHED type or function.

If you want to...	*then see...*	*page*
define the feather descriptor	u$Feather.	9-41
pass a feather to a thread	$tickle.	9-42

u$Feather type

The u$Feather type is a type that must be defined by you. Variables of this type can be used to transfer information to threads such as "terminate processing" or "stop waiting for a signal".

Note: By convention, a thread should terminate itself if it receives a feather pointer that equals $NOADR.

Example: This example defines u$Feather to be a byte. Since a byte has 256 possible representations (0 to 255), a feather of this type could represent 256 different messages when passed into a thread.

```
TYPDEF u$Feather BYTE;
```

$tickle function

$tickle passes a feather to a thread. If the thread is waiting for a signal or a message, it stops waiting and is placed on the ready queue. By convention, a thread should terminate itself if it receives a feather pointer that equals $NOADR.

Note: If the thread is tickled twice before it obtains execution control, the first feather will be lost.

Function Syntax	*Input*	*Output*
$tickle (tcb ^ $ThreadCtlBlk,	Address of the thread control block to tickle.	Thread control block feather field set.
feather ^ u$Feather);	Address of the feather to tickle the thread with.	

Example:

```
killFeather ^ u$Feather := $NOADR;

function () :=
{
   $tickle(tcb1, killFeather);
```

Scheduler logging

Events in the scheduler can be traced by an optional logging facility. If logging is active, each event in the scheduler calls the logging function at the beginning and ending of the scheduler event. Three exceptions are $signal, $sendMsg, and $tickle which only log the beginning of the event.

Variables, types, constants, and functions

This section describes the variables, types, constants, and functions provided for scheduler logging.

Use the following table to select a $SCHED variable, type, constant, or function.

If you want to...	*then see...*	*page*
turn logging on or off	u$eventLoggingOn.	9-44
determine the types of scheduler events logged	$EventType.	9-45
use the mask that signifies the end of an event	$resultOfEvent.	9-46
define the function to log events	u$eventLog.	9-47

u$eventLoggingOn variable

u$eventLoggingOn is a boolean variable that must be declared by you. It is used to determine whether or not to log the events of the scheduler.

Note: u$eventLoggingOn must be declared "ENTRY".

Example:

```
ENTRY u$eventLoggingOn BOOLEAN := TRUE;
```

$EventType type

$EventType contains the values of all of the possible events to be logged by the scheduler.

$EventType is defined as follows:

Type Definition	*Description*
TYPDEF $EventType ENUM ($eventInitSema4,	Initializing a semaphore
$eventFork,	Creating a new thread.
$eventPass,	Thread is giving up execution control.
$eventSwitch,	Scheduling the next thread.
$eventSignal,	Signaling a semaphore.
$eventWaitOn,	Waiting on a semaphore.
$eventSendMsg,	Sending a message.
$eventWaitMsg,	Waiting on a message.
$eventTickle);	Tickling a thread.

$resultOfEvent constant

$resultOfEvent is defined as a constant and is available for you to use.

Constant Definition	*Description*
DEFINE($resultOfEvent, 0x80)	Bit mask that is "or"ed with the $eventTypes logged at the end of an event. This distinguishes the ending log of an event from the beginning log.

Example:

```
ENTRY u$eventLog /* (event $eventType,
                    a1, a2 ^ $String,
                    feather ^ u$feather) */ :=
{
   IF event && $resultOfEvent THEN
      n$format(logStream, 'Event end', LN);
```

u$eventLog function

u$eventLog is a function that must be declared by you if u$eventLoggingOn is TRUE. It should log the events of the scheduler.

Note: u$eventLog must be declared "ENTRY".

Function Syntax	*Input*	*Output*
u$eventLog (event $EventType,	Scheduler event.	
a1 ^ $String,	Address of a string. See table on next page.	
a2 ^ $String,	Address of a string. See table on next page.	
feather ^ u$Feather);	Address of a feather. See table on next page.	

Example:

```
ENTRY u$eventLog /* (event $eventType,
                    a1, a2 ^ $String,
                    feather ^ u$feather) */ :=
{
   IF event && $resultOfEvent THEN
      n$format(logStream, 'Event end.', LN);
   event &&= ~~ $resultOfEvent;
   n$format(logStream, S(events[event]),
            ' ', S(a1^.ptr^, a1^.len));
   IF a2 ~= NIL THEN
      n$format(logStream, ' ', S(a2^.ptr^, a2^.len));
   n$format(logStream, LN);
   };
```

u$eventLog function (continued)

u$eventLog can expect the following input parameters based upon the events listed on the left side of the table:

Event	*a1^*	*a2^*	*feather^*
InitSema4	Semaphore name.	NIL	NIL
Fork	Current thread control block name.	New thread name.	NIL on entry. Current feather on exit.
Pass		NIL	NIL on entry. Feather if tickled on exit.
Switch			NIL on entry. Current feather on exit.
Signal	Semaphore name.		NIL
WaitOn			NIL on entry. Feather if tickled on exit.
SendMsg		Message name.	NIL
WaitMsg		NIL on entry. Message name on exit.	NIL on entry. Feather if tickled on exit.
Tickle	Thread control block name to tickle.	NIL	Feather to tickle with.

Scheduler termination

The threads running under the scheduler are terminated if

- the program ends normally by exiting the MAIN function, or
- an end RMS task is issued such as $ERROR or $EXIT.

$RMSIDLE

Introduction

The $RMSIDLE CUF is the operating system dependent part of the scheduler that includes

- functions that enable a thread to go to sleep while waiting for an RMS I/O to complete, and
- an idle thread which wakes up threads after their I/O has completed.

$RMSIDLE definition file

The $RMSIDLE definition file must be included after the $LINKS and $STRING definition files.

Note: The $RMSIDLE definition file includes the $SCHED definition file.

Example: Your DASL include directives might look like this:

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE($STRING/DEFS)
INCLUDE($LINKS/DEFS)
INCLUDE($RMSIDLE/DEFS)
:
: (rest of the program)
```

$RMSIDLE types and functions

Use the following table to select a $RMSIDLE type or function.

If you want to...	*then see...*	*page*
declare a file access variable descriptor	$FavId.	9-51
start the idle thread	$initRmsIdle.	9-52
wait for I/O to complete	$waitSingleIO.	9-53
give up execution control if there are any waiting threads	$yield.	9-55
declare a function to handle scheduler deadlock	u$deadLock.	9-56
obtain finer control over I/O scheduling	$RMSIDLE low level functions.	9-57

$FavId type

$FavId is defined as follows:

Type Definition	*Description*
TYPDEF $FavId UNSIGNED;	File access variable ID that identifies a file to a thread.

$initRmsIdle function

$initRmsIdle starts up a 0 priority idle thread on the scheduler.

Note: $initRmsIdle should be called right after $initSched is called.

Function Syntax	*Input*	*Output*
$initRmsIdle (privData ^ u$SchedData);	Address of any information that may be unique to this individual thread.	

Example:

```
function () :=
VAR   STATIC name $String := $str('Main Thread');
{
   $initSched(1, &name);
   $initRmsIdle(NIL);
```

$waitSingleIO function

$waitSingleIO informs the idle thread of a pending I/O and causes the current thread to give up execution control while waiting for a specific file I/O to complete.

Function Syntax	*Input*	*Output*
$waitSingleIO (favId $FavId,	File access variable ID of the file with the pending I/O.	
name ^ $String,	Address of a name to be used as the name of the semaphore to wait on.	
completed ^ BOOLEAN,	Address of a boolean.	TRUE if the pending I/O completed. FALSE if tickled before the I/O completed. Note: May be TRUE even if the thread was tickled.
errCode ^ $ERRCODE	Address of a $ERRCODE.	Error code if $SECWAIT error. $CODE field is 0 if no error.
) ^ u$Feather;		Returns the address of a feather if this thread was tickled while asleep.

$waitSingleIO function (continued)

Example:

```
function (pfdb ^ $PFDB) :=
VAR   feather ^ u$Feather;
      STATIC name $String := $str('Readl');
      completed BOOLEAN;
      errCode $ERRCODE;
{
   IF $SECR(pfdb, FALSE) && D$CFLAG THEN $ERMSG();
   feather := $waitSingleIO(pfdb^.$PFVID,
                            &name,
                            &completed,
                            &errCode);
   IF feather ~= NIL THEN
      display('I was tickled awake.');
   IF completed THEN {
      display('The I/O completed.');
      checkForIOError(&errCode);
      };
```

$yield function

$yield causes the thread to give up execution control if there are any threads with higher or equal priorities ready to execute. Otherwise, any threads that were waiting on an I/O are checked to see if their I/O completed. If so, the first thread found is placed on the ready queue and obtains execution control if it has a higher or equal priority than the current thread.

Function Syntax	*Input*	*Output*
$yield ( ) ^ u$Feather;		Returns the address of a feather if this thread was tickled while asleep. NIL if not tickled.

Example:

```
function () :=
{
   IF $yield() ~= NIL THEN
      display('I was tickled awake.')
    ELSE display('The scheduler awakened me.');
```

u$deadLock function

u$deadLock is a function that must be declared by you. It is called if the idle thread obtains execution control and finds

- no threads waiting for an I/O, or
- an I/O that was issued without informing the idle thread by calling $waitSingleIO or $doneIO.

Note: This function must be declared "ENTRY" and should not return to the calling function.

Function Syntax	*Input*	*Output*
u$deadLock (favId $FavId);	NIL if there are no threads waiting on an I/O. Otherwise, a file access variable ID of a file that has a pending I/O that the idle thread was not aware of.	

Example:

```
ENTRY u$deadLock /* (favId $FavId) */ :=
{
    IF favId = NIL THEN display('Scheduler DeadLock')
     ELSE n$format(n$WSOut, 'Unknown FAV ', D(favId));
    $ERROR();
    };
```

$RMSIDLE low level functions

The functions $doneIO, $waitIO, $checkIO, and $stopIO are all low level functions that are called by the $RMSIDLE functions described on the previous pages. In most cases, these functions, along with the $IOMsg type, can be ignored unless finer control is needed.

Low level $RMSIDLE types and functions

Use the following table to select a $RMSIDLE type or function.

If you want to...	*then see...*	*page*
declare an I/O message descriptor	$IOMsg.	9-58
inform the scheduler of an I/O	$doneIO.	9-59
wait for an I/O to complete	$waitIO.	9-60
check for completed I/Os	$checkIO.	9-62
terminate an I/O request	$stopIO.	9-63

$IOMsg type

$IOMsg is defined as follows:

Type Definition	*Description*
TYPDEF $IOMsg STRUCT { msg $Message;	Message used to describe this pending I/O.
compIO ^ $Sema4;	Semaphore to signal once the I/O has completed.
favId $FavId;	File access variable ID of the file with the pending I/O.
errCode $ERRCODE;	Resulting $SECWAIT error if an I/O error occurred.
};	

$doneIO function

$doneIO informs the idle thread that an I/O has been done.

Function Syntax	*Input*	*Output*
$doneIO (msg ^ $IOMsg);	Initialized I/O message. The message name field, the compIO field, and the favId field must be initialized.	

Example:

```
function (pfdb ^ $PFDB) :=
VAR    message $IOMsg;
       STATIC name $String := $str('Readl');
       compIO $Sema4;
{
   IF $SECR(pfdb, FALSE) && D$CFLAG THEN $ERMSG();
   message.msg.name := name;
   message.compIO := &compIO;
   message.favId := pfdb^.$PFVID;
   $doneIO(&message);
```

$waitIO function

$waitIO causes the thread to give up execution control until the I/O announced by the $doneIO completes.

Function Syntax	*Input*	*Output*
$waitIO (compIO ^ $Sema4,	Address of the semaphore to wait on.	
msgPtr ^^ $IOMsg	Address of a pointer to an I/O message.	Points to the I/O message sent by the sending thread. NIL if tickled before receiving the message. Note: This pointer can point to a message even if the thread was tickled.
) ^ u$Feather;		Returns the address of a feather if this thread was tickled while asleep. NIL if not tickled.

$waitIO function (continued)

Example:

```
function (pfdb ^ $PFDB) :=
VAR   message $IOMsg;
      STATIC name $String := $str('Readl');
      compIO $Sema4;
      pMsg ^ $IOMsg;
{
   IF $SECR(pfdb, FALSE) && D$CFLAG THEN $ERMSG();
   message.msg.name := name;
   message.compIO := &compIO;
   message.favId := pfdb^.$PFVID;
   $doneIO(&message);
   $initSema4(&compIO, &name);
   IF $waitIO(&compIO, &pMsg) ~= NIL THEN
      display('I was tickled awake.');
   IF pMsg ~= NIL THEN display('The I/O completed.')
    ELSE display('The I/O did not complete.');
```

$checkIO function

$checkIO checks to see if there are any threads that are waiting on an I/O to see if their I/O completed. If so, the first thread found is placed on the ready queue.

Note: Unlike $yield, execution control is always retained.

Function Syntax	*Input*	*Output*
$checkIO ( ) BOOLEAN;		TRUE if a message was sent to a thread to verify a completed I/O. FALSE otherwise.

Example:

```
function () :=
{
   IF $checkIO() THEN
      display('A thread completed an I/O.');
```

$stopIO function

$stopIO terminates an I/O request that a thread is currently waiting to complete.

Note: A $waitIO must be issued to remove the waiting message from the pending I/O queue.

Function Syntax	*Input*	*Output*
$stopIO (favId UNSIGNED	File access variable ID of the file to stop the I/O.	
) BOOLEAN;		TRUE if the I/O was stopped. FALSE if favId was not waiting on an I/O.

Example:

```
function (pfdb ^ $PFDB) :=
VAR   pMsg ^ $IOMsg;
{
   IF $stopIO(pfdb^.$PFVID) THEN
      display('I/O was terminated.');
   $waitIO(&compIO, &pMsg);
```

Chapter 10.
MISCELLANEOUS CUFS
OVERVIEW

Description

The miscellaneous CUFs are an assortment of CUFs that handle several programming problems that were not covered in the previous chapters.

Selecting a CUF

Use the following table to determine which miscellaneous CUFs are available.

If you want to...	*then use...*	*page*
examine RMS catalogs, subcatalogs, and related files	$catWalk.	10-2
perform mathematical computations on points and rectangles	$EUCLID.	10-6
parse tokens under D$IO with a simple lexical scanner	$LEXER.	10-17
pack or unpack environmental data into and from a character string	$PATH.	10-32
generate a random number (byte)	$RNDBYTE.	10-36

$catWalk

Introduction

The $catWalk CUF is a single function CUF which examines RMS catalogs, subcatalogs, and related files.

$catWalk types and functions

Use the following table to select a $catWalk type or function.

If you want to...	*then see...*	*page*
declare a function as an entry point for $catWalk.	$CatFunc.	10-3
examine a catalog	$catWalk.	10-4

$CatFunc type

Use the $CatFunc type to declare functions that will be called by $catWalk with the information it obtains about the catalogs, subcatalogs, and files.

$CatFunc is defined as follows:

Type Definition	*Description*
TYPDEF $CatFunc (env ^ $ENVT,	A pointer to an environment file entry table.
file ^ $FILEINFO);	A pointer to a file information structure.

Example:

```
function $CatFunc :=
(
```

$catWalk function

$catWalk opens a catalog file and calls a specified function with the result for each valid file in the catalog.

Note: If the function called by $catWalk in turn calls $catWalk, the function must be declared RECURSIVE.

Function Syntax	*Input*	*Output*
$catWalk (openMode BYTE,	Open mode to open the catalog file under. Example: $OMREAD	
env ^ $ENVT,	Address of the environment entry table for the catalog file to examine.	
fileName ^ $NAMET,	Address of the name of the catalog file to examine. Note: If the name is blank, the environment specified by env^ will be used. Otherwise, filename^ must be a catalog under env^.	
func ^ $CatFunc	Address of the function to be called by $catWalk with the result.	Calls func^ with the environment entry table of the current catalog and the filename.
) BOOLEAN;		TRUE if $catWalk completes with no errors. FALSE otherwise.

$catWalk function (continued)

Example: The following program displays all file and catalog names in the :W environment and all subcatalogs of the :W environment.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(D$RMSIO)
INCLUDE(D$UFRENV)
INCLUDE(N$/DEFS)
INCLUDE($catWalk/DEFS)

walk (env ^ $ENVT, file ^ $NAMET);

RECURSIVE write $CatFunc :=
{
   n$format(n$WSOut, S(file^.$FILFNAM), '/',
     S(file^.$FILFEXT), LN);
   IF file^.$FILFEXT = <$EXTT>'' THEN
      walk(env, &file^.$FILFNAM);
   };

RECURSIVE walk :=
{
   IF ~ $catWalk($OMREAD, env, file, &write)
     THEN $ERMSG();
   };

ENTRY MAIN () :=
VAR   env ^^ $ENVT;
      prior ^^ $ENVT;
{
   IF $ENVLOC(&<$ENVN>'W', &env, &prior)
     && D$CFLAG THEN $ERMSG();
   walk(prior^, &<$NAMET>'');
   };
```

$EUCLID

Introduction

The $EUCLID CUF is a euclidean geometry package for performing mathematical computations on points and rectangles.

$EUCLID functions, types, and constants

Use the following table to select a $EUCLID function, type, or constant.

If you want to...	*then see...*	*page*
determine the minimum of two scalars	$MIN.	10-7
determine the maximum of two scalars	$MAX.	10-8
declare a point descriptor	$POINT.	10-9
use one of the predefined points	$EUCLID point constants.	10-9
add two points	$PTADD.	10-10
subtract two points	$PTSUB.	10-11
increment a point	$PTINC.	10-12
decrement a point	$PTDEC.	10-13
declare a rectangle descriptor	$RECTANGLE.	10-14
determine an intersecting rectangle	$INTERSECT.	10-15
determine an enclosing rectangle	$ENCLOSE.	10-16

$MIN function

$MIN determines the minimum of any two DASL scalars.

Function Syntax	*Input*	*Output*
$MIN (num1 Scalar,	First value to be compared.	
num2 Scalar	Second value to be compared.	
) Scalar;		Minimum of the two values.

Example:

```
function (x, y INT) :=
{
   n$format(n$WSOut, 'The minimum of x and y is ',
    D( $MIN(x, y) ), LN);
```

$MAX function

$MAX determines the maximum of any two DASL scalars.

Function Syntax	*Input*	*Output*
$MAX (num1 Scalar,	First value to be compared.	
num2 Scalar	Second value to be compared.	
) Scalar;		Maximum of the two values.

Example:

```
function (x, y INT) :=
{
   n$format(n$WSOut, 'The maximum of x and y is ',
    D( $MAX(x, y) ), LN);
```

$POINT type

$POINT is defined as follows:

Type Definition	*Description*
TYPDEF $POINT STRUCT { x INT;	x axis coordinate.
y INT; };	y axis coordinate.

$EUCLID point constants

The following constants are declared as part of the $EUCLID CUF and are available for you to use.

Constant Definition	*Description*
$PTZERO $POINT := {0, 0};	Point with (0,0) coordinates.
$PTONE $POINT := {1, 1};	Point with (1,1) coordinates.
$PTTWO $POINT := {2, 2};	Point with (2,2) coordinates.

$PTADD function

$PTADD adds the x, y coordinates of one point to another.
a := b + c;

Function Syntax	*Input*	*Output*
$PTADD (a ∧ $POINT,	Address of a point descriptor.	Stores the result of the addition between b∧ and c∧.
b ∧ $POINT,	Address of the first point to add.	
c ∧ $POINT	Address of the second point to add.	
) ∧ $POINT;		Pointer to the input variable a.

Example:

```
function (pPoint1, pPoint2 ∧ $POINT) :=
VAR total $POINT;
{
   $PTADD(&total, pPoint1, pPoint2);
```

$PTSUB function

$PTSUB subtracts the x, y coordinates of one point from another. a := b - c;

Function Syntax	*Input*	*Output*
$PTSUB (a ^ $POINT,	Address of a point descriptor.	Stores the result of the subtraction between b^ and c^.
b ^ $POINT,	Address of the first point to subtract.	
c ^ $POINT	Address of the second point to subtract.	
) ^ $POINT;		Pointer to the input variable a.

Example:

```
function (pPoint1, pPoint2 ^ $POINT) :=
VAR total $POINT;
{
   $PTSUB(&total, pPoint1, pPoint2);
```

$PTINC function

$PTINC increments the x, y coordinates of one point by another. a += b;

Function Syntax	*Input*	*Output*
$PTINC (a ^ $POINT,	Address of the point to increment.	Stores the result of the addition between a^ and b^.
b ^ $POINT	Address of the amount to increment the point by.	
) ^ $POINT;		Pointer to the input variable a.

Example:

```
function (pPoint1, pPoint2 ^ $POINT) :=
{
   $PTINC(pPoint1, pPoint2);
```

$PTDEC function

$PTDEC decrements the x, y coordinates of one point by another. a -= b;

Function Syntax	*Input*	*Output*
$PTDEC (a ^ $POINT,	Address of the point to decrement.	Stores the result of the subtraction between a^ and b^.
b ^ $POINT	Address of the amount to decrement the point by.	
) ^ $POINT;		Pointer to the input variable a.

Example:

```
function (pPoint1, pPoint2 ^ $POINT) :=
{
   $PTDEC(pPoint1, pPoint2);
```

$RECTANGLE type

$RECTANGLE is defined as follows:

Type Definition	*Description*
TYPDEF $RECTANGLE STRUCT { p $POINT;	The bottom left x, y coordinate of the rectangle.
size $POINT; };	Length of the rectangle along the x and y axis.

$INTERSECT function

$INTERSECT determines if there is a common rectangle shared by two rectangles.

Function Syntax	*Input*	*Output*
$INTERSECT (a ^ $RECTANGLE,	Address of a rectangle descriptor.	Stores the values of the common rectangle between b^ and c^.
b ^ $RECTANGLE,	Address of the first rectangle to check for an intersection.	
c ^ $RECTANGLE	Address of the other rectangle to check for an intersection.	
) BOOLEAN;		TRUE if there is a common rectangle. FALSE otherwise.

Example:

```
function (pRec1, pRec2 ^ $RECTANGLE) :=
VAR   interRect $RECTANGLE;
{
   IF $INTERSECT(&interRect, pRec1, pRec2) THEN
      commonRectangle(&interRect);
```

$ENCLOSE function

$ENCLOSE generates the coordinates for a rectangle that encloses two other rectangles.

Function Syntax	*Input*	*Output*
$ENCLOSE (a ^ $RECTANGLE,	Address of a rectangle descriptor.	Stores the values of the rectangle that encloses b^ and c^.
b ^ $RECTANGLE,	Address of the first rectangle to enclose.	
c ^ $RECTANGLE);	Address of the second rectangle to enclose.	

Example:

```
function (pRec1, pRec2 ^ $RECTANGLE) :=
VAR   encloseRect $RECTANGLE;
{
   $ENCLOSE(&encloseRect, pRec1, pRec2);
```

$LEXER

Introduction

The $LEXER CUF is a simple token parsing or lexical scanning package that works under D$IO.

Lexical scanners are most commonly used in compilers where certain characters or grouping of characters, called tokens, are expected in a certain arrangement.

$LEXER definition file

The $LEXER definition file must be included after the $STRING definition file.

Example: Your DASL include directives might look like this:

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE($STRING/DEFS)
INCLUDE($LEXER/DEFS)
:
: (rest of the program)
```

$LEXER types, variables, and functions

Use the following table to select a $LEXER type, variable, or function.

If you want to...	*then see...*	*page*
declare a character class descriptor	$LexClass.	10-19
use a token type	$LexType.	10-20
examine a $LEXER variable	$LEXER variables.	10-21
initialize the character classes	$lexClassInit.	10-22
reclassify a range of characters	$lexClassify.	10-23
reclassify a string of characters	$lexStrClassify.	10-24
initialize the lexical scanner	$lexInit.	10-25
get the next token	$lexNext.	10-26
compare token with a string	$lexEqual.	10-27
terminate if token ~= string	$lexTaste.	10-28
get next token if token = string	$lexEat.	10-29
get next token if token = string otherwise terminate	$lexChew.	10-30
display error and terminate if fatal	$lexError.	10-31

$LexClass type

Every character the lexical scanner parses must be classified to be one of the seven lexical scanner character classes, $LexClass.

$LexClass is defined as follows:

Type Definition	*Description*
TYPDEF $LexClass ENUM ($lexClassAlpha,	Alphabetic characters.
$lexClassDigit,	Numeric characters.
$lexClassSpace,	Space and token division characters.
$lexClassQuote,	Quotation marks.
$lexClassSyntax,	Separators, delimiters, and other syntactic characters.
$lexClassError,	Invalid characters.
$lexClassEnd);	Lexical termination characters.

$LexType type

The character classes are used by the lexical scanner to identify the type of the current token. There are seven lexical scanner token types, $LexType.

$LexType is defined as follows:

Type Definition	*Description*
TYPDEF $LexType ENUM ($lexTypeNil,	No current token type. Note: This value is only used internally by the lexical scanner.
$lexTypeId,	The first character in the current token is a $lexClassAlpha.
$lexTypeNum,	The first character in the current token is a $lexClassDigit.
$lexTypeStr,	The current token is contained within $lexClassQuotes.
$lexTypeSyntax,	The current single character token is a $lexClassSyntax.
$lexTypeEnd);	The current single character token is a $lexClassEnd.

$LEXER variables

The following variables are declared as part of the $LEXER CUF and are available for you to examine.

Variable Definition	*Description*
`$lexLineNo UNSIGNED;`	Current input text file line number.
`$lexErrFlag BOOLEAN;`	TRUE if the lexical scanner error function, $lexError, has been called. FALSE otherwise.
`$curLexType $LexType;`	Lexical scanner type of the current token.
`$curLexVal $String;`	Current token. Note: This value is meaningful only if the token is contained on one line.
`$curLexNum LONG;`	Numeric value of the token. Note: This value is meaningful only if the lexical scanner type of the current token is $lexTypeNum.
`$curLexStr $String;`	String value of the token. Note: This value is meaningful only if the lexical scanner type of the current token is $lexTypeStr.

$lexClassInit function

$lexClassInit sets the character classifications to the predescribed default values.

Function Syntax	*Input*	*Output*
$lexClassInit ( );		

The default character classifications are :

$LexClass Character Classifications	*Default Characters*
$lexClassAlpha	A..Z a..z _ $
$lexClassDigit	0..9
$lexClassSpace	Space $LEOR
$lexClassQuote	' "
$lexClassEnd	$LEOF
$lexClassError	All other characters
$lexClassSyntax	No defaults

Example:

```
function () :=
{
   $lexClassInit();
```

$lexClassify function

$lexClassify reclassifies a range of characters.

Note: This function should not be used to

- reclassify $LEOR, $LEOF or the two quotation marks, or
- classify any other characters as $lexClassQuote.

Function Syntax	*Input*	*Output*
`$lexClassify (` `  from CHAR,`	Beginning character to reclassify.	
`  to CHAR,`	Ending character to reclassify.	
`  lexClass $LexClass` `  );`	Character classification to assign to the characters.	

Example:

```
function () :=
{
   $lexClassInit();
   $lexClassify('a', 'z', $lexClassError);
```

$lexStrClassify function

$lexStrClassify reclassifies all the characters in a string.

Note: This function should not be used to

- reclassify $LEOR, $LEOF or the two quotation marks, or
- classify any other characters as $lexClassQuote.

Function Syntax	*Input*	*Output*
`$lexStrClassify (` `  str [any size] CHAR,`	`Any character string.`	

`  lexClass $LexClass`
 	
`  );`
	`Character classification to assign to the characters.`	

Example:

```
function () :=
{
   $lexClassInit();
   $lexStrClassify('+-*/', $lexClassSyntax);
```

$lexInit function

$lexInit initializes the lexical scanner with the input and error files and scans off the first token.

Note: You must call $lexClassInit prior to calling this function.

Function Syntax	*Input*	*Output*
$lexInit (inFile ^ D$FILET,	Pointer to an open D$IO text file to read.	
errFile ^ D$FILET);	Pointer to an open D$IO text file to write.	

Example:

```
function () :=
{
   $lexClassInit();
   $lexInit(&D$IN, &D$OUT);
```

$lexNext function

$lexNext obtains the next token from the input file.

Function Syntax	*Input*	*Output*
$lexNext ( ) $LexType;		The type of the token obtained from the input file.

Example:

```
function () :=
{
   LOOP WHILE $lexNext() ~= $lexTypeId;
```

$lexEqual function

$lexEqual compares a string to the current token for equality.

Function Syntax	*Input*	*Output*
$lexEqual (str [any size] CHAR	String to compare the current token against.	
) BOOLEAN;		TRUE if the string and token are equal. FALSE otherwise.

Example:

```
function () :=
{
   IF $lexNext() = $lexTypeId
     & $lexEqual('GOOD') THEN goodToken();
```

$lexTaste function

$lexTaste terminates the program if a string and the current token do not match.

Function Syntax	*Input*	*Output*

`$lexTaste (`
`  str [any size] CHAR`
 	
`  );`
	`String to compare the current token against.`	

Example:

```
function () :=
{
   $lexTaste('PASSWORD');
```

$lexEat function

$lexEat obtains the next token if a string and the current token are equal.

Function Syntax	*Input*	*Output*
$lexEat (str [any size] CHAR	String to compare the current token against.	
) BOOLEAN;		TRUE if the string and token are equal. FALSE otherwise.

Example:

```
function () :=
{
   IF ~ $lexEat('PASSWORD') THEN
      $lexError('Password expected', TRUE);
```

$lexChew function

$lexChew terminates the program if a string is not equal to the current token. Otherwise, the next token is obtained.

Function Syntax	*Input*	*Output*
$lexChew (str [any size] CHAR);	String to compare the current token against.	

Example:

```
function () :=
{
   $lexChew('PASSWORD');
```

$lexError function

$lexError displays an error message and terminates on fatal errors.

Function Syntax	*Input*	*Output*
`$lexError (` `  str [any size] CHAR,`	Error message to display.	
`  fatal BOOLEAN` `  );`	Program terminates if TRUE.	

Example:

```
function () :=
{
   IF ~ $lexEat('PASSWORD') THEN {
      $lexError('Password expected', TRUE);
```

$PATH

Introduction

The $PATH CUF packs and unpacks environment data strings that are used by the environment handling routines $ENVINS, $ENVLOC, $catWalk, etc...

$PATH types and functions

Use the following table to select a $PATH type or function.

If you want to...	*then see...*	*page*
declare an environment descriptor	$Path.	10-33
pack environment data into a string	$pathPack.	10-34
unpack environment data	$pathUnPack.	10-35

$Path type

$Path is defined as follows:

Type Definition	*Description*
TYPDEF $Path STRUCT { net $NAMET;	Name of the ARC network for the environment.
node $NAMET;	Name of the node for the environment.
res $NAMET;	Name of the resource for the environment.
hsi $HSI;	Hierarchical structure information for the environment. Note: Information is in standard text format.
nameExt $NAMEEXT;	File name in the environment. Note: This field is not used by either packing or unpacking functions.
nPass BYTE;	Number of passwords for the environment.
passwords [$MAXNPW] $PACKPW; };	Packed passwords for the environment.

$pathPack function

$pathPack takes information in the environment path descriptor and packs it into an environment data string.

Function Syntax	*Input*	*Output*
$pathPack (path ^ $Path,	Address of the environment path descriptor to pack.	
envString ^ CHAR	Address of a character in a character string.	Initialized environment data string.
);		

Example:

```
function () :=
VAR   p $Path;
      ptr ^ CHAR;
      STATIC password $UNPACKPW := 'PASSWORD';
      envDataString [256] CHAR;
{
   p.net  := 'GENESIS     ';
   p.node := 'APD_FP0     ';
   p.res  := 'APD0        ';
   p.hsi  := 'UTILITY                          ';
   p.nPass := 1;
   IF $PAKPW(&password, &p.passwords[0], &ptr)
    && D$CFLAG THEN $ERMSG();
   $pathPack(&p, &envDataString[0]);
```

$pathUnPack function

$pathUnPack takes information in the environment data string and unpacks it into a environment path descriptor.

Function Syntax	*Input*	*Output*
$pathUnPack (envString ^ CHAR,	Address of the the first character in the environment data string to unpack.	
path ^ $Path);	Address of an environment path descriptor.	Initialized environment path descriptor.

Example:

```
function (pEnv ^ CHAR) :=
VAR   p $Path;
{
   $pathUnPack(pEnv, &p);
```

$RNDBYTE

Introduction

The D$RNDBYTE is a single function CUF that generates a random number (byte).

D$RNDBYTE function

D$RNDBYTE generates a random byte.

Function Syntax	*Input*	*Output*
D$RNDBYTE ( ) BYTE;		Generated random number. Note: The generator always starts from the same state.

Example:

```
function () :=
VAR   rand BYTE;
{
   rand := D$RNDBYTE();
```